Scheme and Functional Programming

2006

Preface

This report contains the papers presented at the Seventh Workshop on Scheme and Functional Programming, held on Sunday, September 17, 2006 in Portland, Oregon, affiliated with the 2006 International Conference on Functional Programming (ICFP).

Out of sixteen papers submitted in response to the call for papers, fifteen were chosen for publication by the program committee in a virtual meeting (one was later withdrawn by its author). We are grateful to external reviewers Matthias Blume (Toyota Technological Institute Chicago), Matthew Flatt (University of Utah), Martin Gasbichler (University of Tübingen), Aaron Keen (Cal Poly), George Kuan (University of Chicago), Jacob Matthews (University of Chicago), Christian Queinnec (University Paris 6), and Adam Wick (University of Utah) for their gracious help.

Paper submission and review were managed by the Continue server (`http://continue.cs.brown.edu`), implemented in Scheme, of course. Thanks to Jay McCarthy and Shriram Krishnamurthi who provided and supported the server.

Olin Shivers' writeup of the 2002 Scheme workshop proved invaluable in organizing this one, as did timely advice from Matthias Felleisen, Mike Sperber, and the steering committee. Thanks also to the ICFP general chair John Reppy, the local arrangements chair Jim Hook, and the ACM for handling registrations and arrangements at the workshop site.

Robby Findler
University of Chicago
Organizer and Program Chair
for the program committee

Schedule & Table of Contents

8:30am **Invited Talk: The HOP Development Kit** 7
Manuel Serrano (Inria Sophia Antipolis)

A Stepper for Scheme Macros 15
Ryan Culpepper, Matthias Felleisen (Northeastern University)

10:00am Break

10:30am **An Incremental Approach to Compiler Construction** 27
Abdulaziz Ghuloum (Indiana University)

SHard: a Scheme to Hardware Compiler 39
Xavier Saint-Mleux (Université de Montréal), Marc Feeley (Université de Montréal) and Jean-Pierre David (École Polytechnique de Montréal)

Automatic construction of parse trees for lexemes 51
Danny Dubé (Université Laval) and Anass Kadiri (EPITA, Paris France)

Rapid Case Dispatch in Scheme 63
William D. Clinger (Northeastern University)

Experiences with Scheme in an Electro-Optics Laboratory 71
Richard Cleis and Keith Wilson (Air Force Research Laboratory)

12:30pm Lunch

2:00pm **Gradual Typing for Functional Languages** 81
Jeremy G. Siek and Walid Taha (Rice University)

Sage: Hybrid Checking for Flexible Specifications 93
Jessica Gronski (University of California, Santa Cruz (UCSC)), Kenneth Knowles (UCSC), Aaron Tomb (UCSC), Stephen N. Freund (Williams College), and Cormac Flanagan (UCSC)

From Variadic Functions to Variadic Relations: A miniKanren Perspective 105
William E. Byrd and Daniel P. Friedman (Indiana University)

3:30pm Break

4:00pm **A Self-Hosting Evaluator using HOAS** 119
Eli Barzilay (Northeastern University)

Concurrency Oriented Programming in Termite Scheme 125
Guillaume Germain, Marc Feeley, Stefan Monnier (Université de Montréal)

Interaction-Safe State for the Web 137
Jay McCarthy and Shriram Krishnamurthi (Brown University)

Scheme for Client-Side Scripting in Mobile Web Browsing, or AJAX-Like Behavior Without Javascript 147
Ray Rischpater (Rocket Mobile, Inc.)

Component Deployment with PLaneT: You Want it *Where?* 157
Jacob Matthews (University of Chicago)

6:10pm Break for dinnner
8:00pm Gelato
8:15pm **R^{6}RS Status Report**
Kent Dybvig (Indiana University)

The HOP Development Kit

Manuel Serrano

Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
`http://www.inria.fr/mimosa/Manuel.Serrano`

ABSTRACT

Hop, is a language dedicated to programming reactive and dynamic applications on the web. It is meant for programming applications such as web agendas, web galleries, web mail clients, etc. While a previous paper (*Hop, a Language for Programming the Web 2.0*, available at `http://hop.inria.fr`) focused on the linguistic novelties brought by Hop, the present one focuses on its execution environment. That is, it presents Hop's user libraries, its extensions to the HTML-based standards, and its execution platform, the Hop web *broker*.

DOWNLOAD

Hop is available at: `http://hop.inria.fr`.

The web site contains the distribution of the source code, the online documentation, and various demonstrations.

1. Introduction

Along with games, multimedia applications, and email, the web has popularized computers in everybody's life. The revolution is engaged and we may be at the dawn of a new era of computing where the web is a central element.

Many of the computer programs we write, for professional purposes or for our own needs, are likely to extensively use the web. The web is a database. The web is an API. The web is a novel architecture. Therefore, it needs novel programming languages and novel programming environments. Hop is a step in this direction.

A previous paper [1] presented the Hop programming language. This present paper presents the Hop execution environment. The rest of this section presents the kind of end-user applications Hop focuses on (Section 1.1) and the technical solutions it promotes (Section 1.2). The rest of this paper assumes a familiarity with strict functional languages and with infix parenthetical syntaxes such as the ones found in Lisp and Scheme.

Because it is normal for a web application to access databases, manipulate multimedia documents (images, movies, and music), and parse files according to public formats, programming the web demands a lot of libraries. Even though it is still young, Hop provides many of them. In an attempt to avoid a desperately boring presentation this paper does not present them all! Only the library for building HTML graphical user interfaces is presented here. It is presented in Section 2, along with a presentation of the Hop solution for bringing abstraction to Cascade Style Sheets.

The section 3 focuses on programming the Hop web broker. Its presents basic handling of client requests and it presents the facilities for connecting two brokers and for gathering information scattered on the internet. The Section 4 presents the main functions of the broker programming library.

1.1 The web 2.0

In the newsgroup `comp.lang.functional`, a Usenet news group for computer scientists (if not *researchers* in computer science) someone reacted rather badly to the official announce of the availability of the first version Hop:

"I really don't understand why people are [so] hyped-up over Web 2.0. It's just Java reborn with a slower engine that doesn't even have sandboxing capabilities built into it. I guess this hype will taper off just like the Java hype, leaving us with yet another large technology and a few niches where it's useful."

This message implicitly compares two programming languages, namely Java and JavaScript and reduces Hop to *yet another general-purpose programming language*. This is a misunderstanding. The point of Hop is to help writing new applications that are nearly impossible (or at least, discouragingly tedious) to write using traditional programming languages such as Java and the like. As such, its goal is definitively *not* to compete with these languages.

As a challenge, imagine implementing a program that represents the user with a map of the United States of America that : lets the user zoom in and out on the map, and also helps with trip planning. In particular the user may click on two cities, and the application responds with the shortest route between the cities, the estimated trip time, the price of the gas for the trip (using local pump prices) the weather forecasts along the route (for the appropriate tires), and where to find the best pizza and gelatos in each town along the way. Although it is possible to write such a program using Java or C and existing resources available online, the web 2.0 is the infrastructure that makes it *feasible* to write such programs. Because the web 2.0 provides the potential to easily combine fancy graphics and information from disparate sources online into new, information-aware applications. Unfortunately, the programming model for the web 2.0 is missing. Hop is one attempt to provide the right model, and the rest of this paper explains how.

1.2 The HOP architecture

Hop enforces a programming model where the graphical user interface and the logic of an application are executed on two different engines. In theory, the execution happens as if the two engines are located on different computers even if they are actually frequently hosted by a single computer. In practice, executing a Hop application requires:

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

- A web browser that plays the role of the engine in charge of the graphical user interface. It is the *terminal* of the application. It establishes communications with the Hop *broker*.
- A Hop *broker* which is the execution engine of the application. All computations that involve resources of the local computer (CPU resource, storage devices, various multi-media devices, ...) are executed on the broker. The broker it also in charge of communicating with other Hop brokers or regular web servers in order to gather the information needed by the application.

The Hop programming language provides primitives for managing the distributed computing involved in a whole application. In particular, at the heart of this language, we find the `with-hop` form. Its syntax is:

```
(with-hop (service a0 ..) callback)
```

Informally, its evaluation consists in invoking a remote *service*, i.e., a function hosted by a remote Hop broker, and, on completion, locally invoking the *callback*. The form `with-hop` can be used by engines executing graphical user interfaces in order to spawn computations on the engine in charge of the logic of the application. It can also be used from that engine in order to spawn computations on other remote computation engines.

2. Graphical User Interfaces

This section presents the support of Hop for graphical user interfaces. It presents the library of widgets supported by Hop and its proposal for bringing more abstraction to Cascade Style Sheets (CSS).

2.1 HOP Widgets

Graphical user interfaces are made of elementary graphical objects (generally named *widgets*). Each of these objects has its own graphical aspect and graphical behavior and it reacts to user interactions by intercepting mouse events and keyboard events. Hence, toolkits for implementing graphical user interfaces are characterized by:

1. the mechanisms for catching user interactions, and
2. the composition of graphical elements, and
3. the richness of them widgets.

HTML (either W3C's HTML-4 or XHTML-1) do a good job at handling events. Each HTML elements is *reactive* and JavaScript, the language used for programming events handlers, is adequate. CSS2, the HTML composition model based on boxes, is close to be sufficient. The few lacking facilities are up to be added to the third revision. On the other hand, the set of HTML widgets is poor. It mainly consists of boxes, texts, and buttons. This is insufficient if the web is considered for implementing modern graphical user interfaces. Indeed, these frequently use *sliders* for selecting integer values, *trees* for representing recursive data structures, *notepads* for compact representations of unrelated documents, and many others. HTML does not support these widgets and, even worse, since it is not a programming language, it does not allow user to implement their own complementary sets of widgets. Hop bridges this gap.

Hop proposes a set of widgets for easing the programming of graphical user interfaces. In particular, it proposes a *slider* widget for representing numerical values or enumerated sets. It proposes a WYSIWYG editor. It extends HTML tables for allowing automatic sorting of columns. It supports various *container* widgets such as a *pan* for splitting the screen in two horizontal or vertical re-sizable areas, a *notepad* widget for implementing *tab* elements, a *hop-iwindow* that implements a window system in the browser, etc. In this paper, we focus on one widget that is representative of the container family, the *tree* widget.

2.1.1 The tree widget

A tree is a traditional widget that is frequently used for representing its eponymous data structure. For instance, it is extensively used for implementing file selectors. The syntax of Hop trees is given below. The meta elements required by the syntax are expressed using lower case letters and prefixed with the character `%`. The concrete markups only use upper case letters. The meta element `%markup` refers to the whole set of Hop markups.

```
%markup ⟶ ... | %tree

%tree ⟶ (<TREE> %tree-head %tree-body)
%tree-head ⟶ (<TRHEAD> %markup)
%tree-body ⟶ (<TRBODY> %leaf-or-tree*)
%leaf-or-tree ⟶ %leaf | %tree
%leaf ⟶ (<TRLEAF> %markup)
```

As an example, here is a simple tree.

```
(define (dir->tree dir)
  (<TREE>
    (<TRHEAD> dir)
    (<TRBODY>
      (map (lambda (f)
             (let ((p (make-file-name dir f)))
               (if (directory? p)
                   (dir->tree p)
                   (<TRLEAF> :value qf f))))
           (directory->list dir)))))
```

When an expression such as `(dir->tree "/")` is evaluated on the broker, a tree widget representing the hierarchy of the broker files is built. It has to be sent to a client for rendering.

Hop containers (i.e., widgets that contain other widgets) are *static*, as in the example above, or *dynamic*. A static container builds its content only once. A dynamic container rebuilds its content each time it has to be displayed. A static tree has a fixed set of subtrees and leaves. A dynamic tree recomputes them each time unfolded. A dynamic tree is characterized by the use of the `<DELAY>` markup in its body. The syntax of this new markup is:

```
(<DELAY> thunk)
```

The argument `thunk` is a procedure of no argument. Evaluating a `<DELAY>` form on the Hop broker installs an anonymous service whose body is the application of this `thunk`. When the client, i.e., a web browser, unfolds a dynamic tree, its invokes the service associated with the `thunk` on the broker. This produces a new tree that is sent back to the client and inserted in the initial tree.

```
(define (dir->dyntree dir)
  (<TREE>
    (<TRHEAD> dir)
    (<TRBODY>
      (<DELAY>
        (lambda ()
          (map (lambda (f)
                 (let ((p (make-file-name dir f)))
                   (if (directory? p)
                       (dir->dyntree p)
                       (<TRLEAF> :value qf f))))
               (directory->list dir)))))))
```

Even if the function `dir->dyntree` only differs from `dir->tree` by the use of the `<DELAY>` markup, its execution is dramatically different. When the expression `(dir->dyntree "/")` is evaluated, the broker no longer traverses its entire hierarchy of files. It only

inspects the files located in the directory `"/"`. When the client, i.e., a web browser, unfolds a node representing a directory, the broker traverses only that directory for scanning the files. Contrary to `dir->tree`, the directories associated with nodes that are never unfolded are never scanned by `dir->dyntree`.

2.1.2 Extending existing HTML markups

Because Hop is not HTML it is very tempting to add some HTML facilities to Hop, for instance by adding new attributes to markups. In order to keep the learning curve as low as possible, we resist this temptation. Hop offers the HTML markups as is, with on exception: the `<IMG>` markup. In HTML, this markup has a `src` attribute that specifies the actual implementation of the image. It can be an URL or an in-line encoding of the image. In that case, the image is represented by a string whose first part is the declaration of a mime type and the second part a row sequence of characters representing the encoding (e.g., a *base64* encoding of the bytes of the image). While this representation is close to impractical for a hand-written HTML documents, it is easy to produce for automatically generated documents, such as the ones produced by Hop. Hop adds a new attribute `inline` to HTML images. When this attribute is set to `#t` (the representation of the value *true* in the concrete Hop syntax), the image is encoded on the fly.

This tiny modification to HTML illustrates why a programming language can dramatically help releasing documents to the web. Thanks to this `inline` attribute, it is now easy to produce stand alone HTML files. This eliminates the burden of packaging HTML documents with external tools such as `tar` or `zip`.

2.2 HOP Cascade Style Sheets

Cascading Style Sheets (CSS) enable graphical customizations of HTML documents. A CSS specifies rendering information for visualizing HTML documents on computer screens, printing them on paper, or even pronouncing them on aural devices. A CSS uses selectors to designate the elements onto which a customization applies. Attributes, which are associated with selectors, specify the rendering information. The set of possible rendering attributes is rich. CSS exposes layout principles based on horizontal and vertical boxes in the spirit of traditional text processing applications. CSS version 2 suffers limitations (for instance, it only supports one column layout) that are to be overcome by CSS version 3. CSS is so expressive that we think that when CSS v3 is fully supported by web browsers, HTML will compete with text processors like Latex for printing high quality documents.

CSS selectors are expressed in a little language. The elements to which a rendering attribute applies are designed either by their identities, their classes, their local or global positions in the HTML tree, and their attributes. The language of selectors is expressive but complex, even if not Turing-complete. On the one hand, the identity and class designations are suggestive of object-oriented programming. On the other hand, they do not support inheritance. Implementing re-usable, compact, and easy-to-understand CSS is a challenging task. Frequently the HTML documents have to be modified in order to best fit the CSS model. For instance, dummy `<DIV>` or `<SPAN>` HTML elements have to be introduced in order to ease the CSS selection specification. We think that this complexity is a drawback of CSS, and Hop offers an improvement.

Like the Hop programming language, Hop-CSS (HSS in short) uses a stratified language approach. HSS extends CSS in one direction: it enables embedding, inside standard CSS specifications, Hop expressions. The CSS syntax is extended with a new construction. Inside a HSS specification, the `$` character escapes from CSS and switches to Hop. This simple stratification enables arbitrary Hop expressions to be embedded in CSS specifications. We have found this extension to be useful to avoiding repeating constants. For instance, instead of duplicating a color specification in many attributes, it is convenient to declare a variable holding the color value and use that variable in the CSS. That is, the traditional CSS:

```
button {
  border: 2px inset #555;
}
span.button {
  border: 2px inset #555;
}
```

in Hop can be re-written as:

```
$(define border-button-spec "2px inset #555")

button {
  border: $border-button-spec;
}
span.button {
  border: $border-button-spec;
}
```

In other situations, the computation power of Hop significantly helps the CSS specifications. As an example, imagine a graphical specification for 3-dimensional borders. Given a base color, a 3-dimensional inset border is implemented by lightening the top and left borders and darkening the bottom and right borders. Using the two Hop library functions `color-ligher` and `color-darker` this can be implemented as:

```
$(define base-color "#555")

button {
  border-top: 1px solid $(color-lighter base-color);
  border-left: 1px solid $(color-lighter base-color);
  border-bottom: 1px solid $(color-darker base-color);
  border-right: 1px solid $(color-darker base-color);
}
```

The specification of the buttons border is actually a compound property made of four attributes. It might be convenient to bind these four attributes to a unique Hop variable. Since the HSS `$` escape character enables to inject compound expressions, this can be wriiten as:

```
$(define base-color "#555")
$(define button-border
   (let ((c1 (color-lighter base-color))
         (c2 (color-darker base-color)))
      { border-top: 1px solid $c1;
        border-left: 1px solid $c2;
        border-bottom: 1px solid $c2;
        border-right: 1px solid $c1 }))

button {
   $button-border;
}
```

3. Programming the HOP web broker

The Hop web broker implements the execution engine of an application. While the client executes in a sandbox, the broker has privileged accesses to the resources of the computer it execution on. As a consequence, the client has to delegate to the broker the operations it is not allowed to execute by itself. These operations might be reading a file, executing a CPU-intensive operation, or collecting information from another remote Hop broker or from a remote web server. In that respect, a Hop broker is more than a web server because it may act has a client itself for handling external requests. Still, a Hop broker resembles a web server. In particular, it conforms to the HTTP protocol for handling clients connections

and requests. When a client request is parsed, the broker elaborates a response. This process is described in the next sections.

3.1 Requests to Responses

Clients send HTTP messages to Hop brokers that parse the messages and build objects representing these requests. For each such objects, a broker elaborates a response. Programming a broker means adding new rules for constructing responses. These rules are implemented as functions accepting requests. On return, they either produce a new request or a response. The algorithm for constructing the responses associated with requests is defined as follows.

```
(define (request->response req rules)
   (if (null? rules)
       (default-response-rule req)
       (let ((n ((car rules) req)))
          (cond
             ((is-response? n)
              n)
             ((is-request? n)
              (request->response n (cdr rules)))
             (else
              (request->response req (cdr rules)))))))
```

The *else* branch of the conditional is used when no rule applies. It allows rules to be built using `when` and `unless`, without having to be a series of nested `if`s.

A rule may produce a response. In that case, the algorithm returns that value. A rule may also annotate a request or build a new request from the original one. In that case, the algorithm applies the remaining rules to that new request.

The default response rule, which is used when no other rule matches, is specified in the configuration files of the broker.

3.2 Producing responses

The broker has to serve various kind of responses. Some responses involve local operations (such as serving a file located on the disk of the computer where the broker executes). Some other responses involve fetching information from the internet. Hop proposes several type of responses that correspond to the various ways it may fulfill client requests.

From a programmer's point of view, responses are represented by subclasses of the abstract class `%http-response`. Hop proposes an extensive set of pre-declared response classes. The most important ones are presented in the rest of this section. Of course, user programs may also provide new response classes.

3.2.1 No response!

Responses instance of the class `http-response-abort` are actually *no response*. These objects are used to prevent the broker for answering unauthorized accesses. For instance, on may wish to prevent the broker for serving requests originated from a remote host. For that, he should had a rule that returns an instance of `http-response-abort` for such requests.

Hop provides predicates that return true if and only if a request comes from the local host. Hence, implementing remote host access restriction can be programmed as follows.

```
(hop-add-rule!
 (lambda (req)
    (if (is-request-local? req)
        req
        (instantiate::http-response-abort))))
```

3.2.2 Serving files

The class `http-response-file` is used for responding files. It is used for serving requests that involve static documents (static HTML documents, cascade style sheets, etc.). It declares the field `path` which is used to denote the file to be served. In general these responses are produced by rules equivalent to the following one.

```
(hop-add-rule!
 (lambda (req)
    (if (and (is-request-local? req)
             (file-exists? (request-path req)))
        (instantiate::http-response-file
           (path (request-path req))))))
```

In order to serve `http-response-file` responses, the broker reads the characters from the disk and transmit them to the client via a socket. Some operating systems (such as Linux 2.4 and higher) propose system calls for implementing this operation efficiently. This liberates the application from explicitly reading and writing the characters of the file. With exactly one system call, the whole file is read and written to a socket. For this, Hop uses subclasses of `http-response-file`.

The class `http-response-shoutcast` is one of them. It is used for serving music files according to the shoutcast protocol[1]. This protocol adds meta-information such as the name of the music, the author, etc., to the music broadcasting. When a client is ready for receiving shoutcast information, it must add an `icy-metadata` attribute to the header of its requests. Hence, in order to activate shoutcasting on the broker one may use a rule similar to the following one.

```
(hop-add-rule!
 (lambda (req)
    (if (and (is-request-local? req)
             (file-exists? (request-path req)))
        (if (is-request-header? req 'icy-metadata)
            (instantiate::http-response-shoutcast
               (path (request-path req)))
            (instantiate::http-response-file
               (path (request-path req)))))))
```

Note that since the rules scanned in the inverse order of the their declaration, the shoutcast rule must be added after the rule for regular files.

3.2.3 Serving dynamic content

Hop provides several classes for serving dynamic content. The first one, `http-response-procedure`, is used for sending content that varies for each request. The instances of that class carry a procedure that is invoked each time the response is served. In the example above, we add a rule that create a *virtual* URL `/count` that returns the value of an incremented counter each time visited.

```
(let ((count 0)
      (resp (instantiate::http-response-procedure
               (proc (lambda (op)
                        (set! count (+ 1 count))
                        (printf op
                                "<HTML>~a</HTML>"
                                count))))))
   (hop-add-rule!
    (lambda (req)
       (when (and (is-request-local? req)
                  (string=? (request-path req) "/count"))
          resp))))
```

3.2.4 Serving data

Hop programs construct HTML documents on the server. On demand they are served to clients. These responses are implemented

[1] `http://www.shoutcast.com/`.

by the http-response-hop class. When served, the XML tree inside a response of this type is traversed and sent to the client. As an example, consider a rule that adds the URL /fact to the broker. That rule computes a HTML table filled with factorial numbers.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (string=? (request-path req) "/fact"))
       (instantiate::http-response-hop
          (xml (<TABLE>
                  (map (lambda (n)
                          (<TR>
                             (<TH> n)
                             (<TD> (fact n))))
                       (iota 10 1))))))))
```

Instead of always computing factorial value from 1 to 10, it is easy to modify the rule for adding a range.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (substring? (request-path req) "/fact/"))
       (let ((m (string->integer
                 (basename (request-path req)))))
          (instantiate::http-response-hop
             (xml (<TABLE>
                     (map (lambda (n)
                             (<TR>
                                (<TH> n)
                                (<TD> (fact n))))
                          (iota m 1)))))))))
```

Next, we now show how to modify the rule above so that the computation of the HTML representation of the factorial table is moved from the broker to the client. As presented in Section 1.2, the Hop programming language supports the form with-hop. This invokes a service on the broker and applies, on the client, a callback with the value produced by the service. This value might be an HTML fragment or another Hop value. On the server, HTML fragments are represented by responses of the class http-response-hop. The other values are represented by the class http-response-js. When such a response is served to the client, the value is serialized on the broker according to the JSON format[2] and unserialized on the client. We can re-write the previous factorial example in order to move the computation of the HTML table from the broker to the client. For that, we create a rule that returns the factorial values in a list.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (substring? (request-path req) "/fact/"))
       (let ((m (string->integer
                 (basename (request-path req)))))
          (instantiate::http-response-js
             (value (map (lambda (n)
                            (cons n (fact n)))
                         (iota m 1))))))))
```

The /fact URL can be used in client code as follows.

```
(with-hop "/hop/fact/10"
   (lambda (l)
      (<TABLE>
         (map (lambda (p)
                 (<TR>
                    (<TH> (car p))
                    (<TD> (cdr p))))
              l))))
```

The point of this last example is not to argue in favor of moving this particular computation from the broker to the client. It is just to show how these moves can be programmed with Hop.

3.2.5 Serving remote documents

Hop can also act as a web proxy. In that case, it intercepts requests for remote hosts with which it establishes connections. It reads the data from those hosts and sends them back to its clients. The class http-response-remote represents such a request.

In order to let Hop act as a proxy, one simply adds a rule similar to the one below.

```
(hop-add-rule!
 (lambda (req)
    (unless (is-request-local? req)
       (instantiate::http-response-remote
          (host (request-host req))
          (port (request-port req))
          (path (request-path req))))))
```

This rule is a good candidate for acting as the *default* rule presented in Section 3.1. The actual Hop distribution uses a default rule almost similar to this one. It only differs from this code by returning an instance of the http-response-string class for denoting a *404 error* when the requests refer to local files.

3.2.6 Serving strings of characters

Some requests call for simple responses. For instance when a request refers to an non existing resource, a simple error code must be served to the client. The class http-response-string plays this role. It is used to send a return code and, optionally, a message, back to the client.

The example below uses a http-response-string to redirect a client. From time to time, Google uses *bouncing* which is a technique that allows them to log requests. That is, when Google serves a request, instead of returning a list of found URLs, it returns a list of URLs pointing to Google, each of these URL containing a forward pointer to the actual URL. Hence Google links look like:

```
http://www.google.com/url?q=www.inria.fr
```

When Hop is configured for acting as a proxy it can be used to avoid this bouncing. A simple rule may redirect the client to the actual URL.

```
(hop-add-rule!
 (lambda (req)
    (when (and (string=? (request-host req)
                         "www.google.com")
               (substring? (request-path req) "/url" 0))
       (let ((q (cgi-fetch-arg "q" path)))
          (instantiate::http-response-string
             (start-line "HTTP/1.0 301 Moved Permanently")
             (header (list (cons 'location: q))))))))
```

A similar technique can be used for implementing blacklisting. When configured as web proxy, Hop can be used to ban ads contained in HTML pages. For this, let us assume a black list of domain names held in a hash table loaded on the broker. The rule

[2] http://www.json.org/.

above prevents pages from these domains to be served. It lets the client believe that ads pages do not exist.

```
(hop-add-rule!
 (lambda (req)
    (when (hashtable-get *blacklist* (request-host req))
       (instantiate::http-response-string
          (start-line "HTTP/1.0 404 Not Found")))))
```

3.3 Broker hooks

When a response is generated by the algorithm presented in Section 3.1 and using the rules of Section 3.2 the broker is ready to fulfill a client request. Prior to sending the characters composing the answer, the broker still offers an opportunity to apply programmable actions to the generated request. That is, before sending the response, the broker applies *hooks*. A hook is a function that might be used for applying security checks, for authenticating requests or for logging transactions.

A hook is a procedure of two arguments: a request and a response. It may modify the response (for instance, for adding extra header fields) or it may return a new response. In the following example, a hook is used to restrict the access of the files of the directory `/tmp`.

```
(hop-hook-add!
 (lambda (req resp)
  (if (substring? (request-path req) "/tmp/")
   (let ((auth (get-request-header req 'authorization)))
    (if (file-access-denied? auth "/tmp")
     (instantiate::http-response-authentication
       (header '("WWW-Authenticate: Basic realm=Hop"))
       (body (format "Authentication required.")))
     resp))
   resp)))
```

When a request refers to a file located in the directory `/tmp`, the hook presented above forces Hop to check if that request is authenticated (a request is authenticated when it contains a header field `authorization` with correct values). When the authentication succeeds, the file is served. Otherwise, a request for authentication is sent back to the client.

4. The HOP library

The Hop standard library provides APIs for graphical user interfaces, for enabling communication between the clients and the broker, for decoding standards documents formats (e.g., EXIF for `jpeg` pictures, ID3 for `mp3` music, XML, HTML, RSS, ...). It also offers APIs for enabling communications between two brokers and between brokers and regular web servers. Since the communication between two brokers is similar to the communication between clients and brokers (see the form `with-hop` presented Section 1.2), it is not presented here. In this section we focus on the communications between brokers and regular web servers.

The Hop library provides facilities for dealing with low-level network communications by the means of sockets. While this is powerful and enables all kind of communications it is generally tedious to use. In order to remove this burden from programmers, Hop provides two high-level constructions: the `<INLINE>` markup and the `with-url` form.

4.1 The `<INLINE>` markup

The `<INLINE>` markup lets a document embed subparts of another remote document. When the broker sends a HTML tree to a client, it *resolves* its `<INLINE>` nodes. That is, it opens communication with the remote hosts denoted to by the `<INLINE>` nodes, it parses the received documents and it includes these subtrees to the response sent to its client.

The `<INLINE>` node accepts two options. The first one, `:src`, is mandatory. It specifies the URL of the remote host. The example below builds a HTML tree reporting information about the current version of the Linux kernel. This information is fetched directly from the kernel home page. It is contained in an element whose identifier is `versions`.

```
(<HTML>
  (<BODY>
    "The current Linux kernel versions are:"
    (let ((d (<INLINE> :src "http://www.kernel.org")))
       (dom-get-element-by-id d "versions"))))
```

This program fetches the entire kernel home page. From that document it extracts the node named `versions`. The second option of the `<INLINE>` node allows a simplification of the code by automatically isolating one node of the remote document. The `:id` option restricts the inclusion, inside the client response, to one element whose identifier is `:id`. Using this second option, our program can be simplified as shown below.

```
(<HTML>
  (<BODY>
     "The current Linux kernel versions are:"
     (<INLINE> :src "http://www.kernel.org"
               :id "versions")))
```

In addition to be more compact, this version is also more efficient because it does not require the entire remote document to be loaded on the broker. As it receives characters from the network connection, the broker parses the document. As soon as it has parsed a node whose identifier is `versions` it closes the connection.

4.2 The `with-url` form

The syntax of the form `with-url` is as follows:

```
(with-url url callback)
```

Informally, its evaluation consists in fetching a remote document from the web and on completion, invoking the *`callback`* with the read characters as argument. Unlike to the `<INLINE>` node, the characters do not need to conform any particular syntax. More precisely, the fetched document does not necessarily need to be a valid XML document. In the example below, we show how the `with-url` form can be used to implement a simple RSS reader.

The function `rss-parse` provided by the standard Hop library parses a string of characters according to the RSS grammar. It accepts four arguments, the string to be parsed and three constructors. The first and seconds build a data structure representing RSS sections. The last one builds data structures for representing RSS entries.

```
(define (make-rss channel items)
   (<TREE>
      channel
      (<TRBODY> items)))

(define (make-channel channel)
   (<TRHEAD> channel))
```

```
(define (make-item link title date subject descr)
   (<TRLEAF>
      (<DIV>
         :class "entry"
         (<A> :href link title)
         (if date (list "(" date ")"))
         (if subject (<I> subject))
         descr)))
```

Once provided with the tree constructors, parsing RSS documents is straightforward.

```
(define (rss->html url)
   (with-url url
      (lambda (h)
         (rss-parse h make-rss make-channel make-item))))
```

Producing a RSS report is then as simple as:

```
(rss->html "kernel.org/kdist/rss.xml")
```

5. Conclusion

Hop is a programming language dedicated to programming interactive web applications. It differs from general purpose programming languages by providing support for dealing with programs whose execution is split across two computers. One computer is in charge of executing the logic of the application. The other one is in charge of dealing with the interaction with users.

This article focuses on the Hop development kit. It presents some extensions to HTML that enable fancy graphical user interfaces programming and it presents the Hop web broker programming. In the presentation various examples are presented. In particular, the paper shows how to implement simple a RSS reader with Hop in no more than 20 lines of code!

The Hop library is still missing important features for web programming. In particular, it does not provide SOAP interface, it cannot handle secure HTTPS connections, and it does not implement graphical visual effects. We continue to work on Hop, however, and would love your feedback.

6. References

[1] Serrano, et al. – **Hop, a Language for Pogramming the Web 2.0** – 2006.

Acknowledgments

I would like to thanks Robby Findler for his invitation to the Scheme workshop and for his extremely helpful comments on the paper.

A Stepper for Scheme Macros

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Even in the days of Lisp's simple `defmacro` systems, macro developers did not have adequate debugging support from their programming environment. Modern Scheme macro expanders are more complex than Lisp's, implementing lexical hygiene, referential transparency for macro definitions, and frequently source properties. Scheme implementations, however, have only adopted Lisp's inadequate macro inspection tools. Unfortunately, these tools rely on a naive model of the expansion process, thus leaving a gap between Scheme's complex mode of expansion and what the programmer sees.

In this paper, we present a macro debugger with full support for modern Scheme macros. To construct the debugger, we have extended the macro expander so that it issues a series of expansion events. A parser turns these event streams into derivations in a natural semantics for macro expansion. From these derivations, the debugger extracts a reduction-sequence (stepping) view of the expansion. A programmer can specify with simple policies which parts of a derivation to omit and which parts to show. Last but not least, the debugger includes a syntax browser that graphically displays the various pieces of information that the expander attaches to syntactic tokens.

1. The Power of Macros

Modern functional programming languages support a variety of abstraction mechanisms: higher-order functions, expressive type systems, module systems, and more. With functions, types, and modules, programmers can develop code for reuse; establish single points of control for a piece of functionality; decouple distinct components and work on them separately; and so on. As Paul Hudak [18] has argued, however, "the ultimate abstraction of an application is a . . . language." Put differently, the ideal programming language should allow programmers to develop and embed entire sub-languages.

The Lisp and Scheme family of languages empower programmers to do just that. Through macros, they offer the programmer the ability to define *syntactic abstractions* that manipulate binding structure, perform some analyses, re-order the evaluation of expressions, and generally transform syntax in complex ways—all at compile time. As some Scheme implementors have put it, macros have become a true compiler (front-end) API.

In the context of an expressive language [9] macros suffice to implement many general-purpose abstraction mechanisms as libraries that are *indistinguishable from built-in features*. For example, programmers have used macros to extend Scheme with constructs for pattern matching [32], relations in the spirit of Prolog [8, 27, 15, 20], extensible looping constructs [7, 26], class systems [24, 1, 14] and component systems [30, 13, 5], among others. In addition, programmers have also used macros to handle tasks traditionally implemented as *external* metaprogramming tasks using preprocessors or special compilers: Owens et al. [23] have added a parser generator library to Scheme; Sarkar et al. [25] have created an infrastructure for expressing nano-compiler passes; and Herman and Meunier [17] have used macros to improve the set-based analysis of Scheme. As a result, implementations of Scheme such as PLT Scheme [12] have a core of a dozen or so constructs but appear to implement a language the size of Common Lisp.

To support these increasingly ambitious applications, macro systems had to evolve, too. In Lisp systems, macros are compile-time functions over program fragments, usually plain S-expressions. Unfortunately, these naive macros don't really define abstractions. For example, these macros interfere with the lexical scope of their host programs, revealing implementation details instead of encapsulating them. In response, Kohlbecker et al. [21] followed by others [4, 6, 11] developed the notions of macro hygiene, referential transparency, and phase separation. In this world, macros manipulate syntax tokens that come with information about lexical scope; affecting scope now takes a deliberate effort and becomes a part of the macro's specification. As a natural generalization, modern Scheme macros don't manipulate S-expressions at all but opaque syntax representations that carry additional information. In the beginning, this information was limited to binding information; later Dybvig et al. [6] included source information. Now, Scheme macros contain arbitrary properties [12] and programmers discover novel uses of this mechanism all the time.

Although all these additions were necessary to create true syntactic abstraction mechanisms, they also dramatically increased the complexity of macro systems. The result is that both inexperienced and advanced users routinely ask on Scheme mailing lists about unforeseen effects, subtle errors, or other seemingly inexplicable phenomena. While "macrologists" always love to come to their aid, these questions demonstrate the need for software tools that help programmers explore their macro programs.

In this paper, we present the first macro stepper and debugger. Constructing this tool proved surprisingly complex. The purpose of the next section is to explain the difficulties abstractly, before we demonstrate how our tool works and how it is constructed.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

2. Explaining Macros

Macro expansion takes place during parsing. As the parser traverses the concrete syntax,[1] it creates abstract syntax nodes for primitive syntactic forms, but stops when it recognizes the use of a macro. At that point, it hands over the (sub)phrases to the macro, which, roughly speaking, acts as a rewriting rule.

In Lisp and in some Scheme implementations, a macro is expressed as a plain function; in R^{5}RS Scheme [19], macros are expressed in a sub-language of rewriting rules based on patterns. Also in Lisp, concrete syntax are just S-expressions; Lisp macro programming is thus typically first-order functional programming[2] over pairs and symbols. The most widely used Scheme implementations, however, represent concrete syntax with structures that carry additional information: lexical binding information, original source location, code security annotations, and others. Scheme macro programming is therefore functional programming with a rich algebraic datatype.

Given appropriate inputs, a Lisp macro can go wrong in two ways. First, the macro transformer itself may raise a run-time exception. This case is naturally in the domain of run-time debuggers; after all, it is just a matter of traditional functional programming. Second, a Lisp macro may create a new term that misuses a syntactic form, which might be a primitive form or another macro. This kind of error is not detected when the macro is executing, but only afterwards when the parser-expander reaches the misused term.

Modern Scheme macros might go wrong in yet another way. The additional information in a syntax object interacts with other macros and primitive special forms. For example, macro-introduced identifiers carry a mark that identifies the point of their introduction and binding forms interpret identifiers with different marks as distinct names. Scheme macros must not only compute a correct replacement tree but also equip it with the proper additional properties.

Even in Lisp, which has supported macros for almost 50 years now, macros have always had impoverished debugging environments. A typical Lisp environment supports just two procedures/tools for this purpose: `expand` and `expand-once` (or `macroexpand` and `macroexpand-1` [28]). All Scheme implementations with macros have adapted these procedures.

When applied to a term, `expand` completely parses and expands it; in particular, it does not show the intermediate steps of the rewriting process. As a result, `expand` distracts the programmer with too many irrelevant details. For example, Scheme has three conditional expressions: `if`, `cond`, and `case`. Most Scheme implementations implement only `if` as a primitive form and define `cond` and `case` as macros. Whether or not a special form is a primitive form or a macro is irrelevant to a programmer except that macro expansion reveals the difference. It is thus impossible to study the effects of a single macro or a group of related macros in an expansion, because `expand` processes *all* macros and displays the entire abstract syntax tree.

The task of showing individual expansion steps is left to the second tool: `expand-once`. It consumes a macro application, applies the matching macro transformer, and returns the result. In particular, when an error shows up due to complex macro interactions, it becomes difficult to use `expand-once` easily because the offending or interesting pieces are often hidden under a large pile of syntax. Worse, iterated calls to `expand-once` lose information between expansion steps, because lexical scope and other information depends on the context of the expansion call. This problem renders `expand-once` unfit for serious macro debugging.

[1] We consider the result of `(read)` as syntax.

[2] Both Lisp and Scheme macro programmers occasionally use side-effects but aside from `gensym` it is rare.

Implementing a better set of debugging tools than `expand` and `expand-once` is surprisingly difficult. It is apparently impossible to adapt the techniques known from run-time debugging. For example, any attempt to pre-process the syntax and attach debugging information or insert debugging statements fails for two reasons: first, until parsing and macro expansion happens, the syntactic structure of the tree is unknown; second, because macros inspect their arguments, annotations or modifications are likely to change the result of the expansion process [31].

While these reasons explain the dearth of macro debugging tools and steppers, they don't reduce the need for them. What we present in this paper is a mechanism for instrumenting the macro expander and for displaying the expansion events and intermediate stages in a useful manner. Eventually we also hope to derive a well-founded model of macros from this work.

3. The Macro Debugger at Work

The core of our macro debugging tool is a stepper for macro expansion in PLT Scheme. Our macro debugger shows the macro expansion process as a reduction sequence, where the redexes are macro applications and the contexts are primitive syntactic forms, i.e., nodes in the final abstract syntax tree. The debugger also includes a syntax display and browser that helps programmers visualize properties of syntax objects.

The macro stepper is parameterized over a set of "opaque" syntactic forms. Typically this set includes those macros imported from libraries or other modules. The macro programmers are in charge, however, and may designate macros as opaque as needed. When the debugger encounters an opaque macro, it deals with the macro as if it were a primitive syntactic form. That is, it creates an abstract syntax node that hides the actual expansion of the macro. Naturally, it does show the expansion of the subexpressions of the macro form. The parameterization of primitive forms thus allows programmers to work at the abstraction level of their choice. We have found this feature of the debugger critical for dealing with any nontrivial programs.

The rest of this section is a brief illustrative demonstration of the debugger. We have picked three problems with macros from recent discussions on PLT Scheme mailing lists, though we have distilled them into a shape that is suitably simple for a technical paper.

3.1 Plain Macros

For our first example we consider a debugging scenario where the macro writer gets the form of the result wrong. Here are three different versions of a sample macro that consumes a list of identifiers and produces a list of trivial definitions for these identifiers:

1. in Lisp, the macro writer uses plain list-processing functions to create the result term:

```
(define-macro (def-false . names)
  (map (lambda (a) `(define ,a #f)) names))
```

2. in R^{5}RS the same macro is expressed with a rewriting rule notation like this:

```
(define-syntax def-false
  (syntax-rules ()
    [(def-false a ...) ((define a #f) ...)]))
```

3. in major alternative Scheme macro systems, the rule specification is slightly different:

```
(define-syntax (def-false stx)
  (syntax-case stx ()
    [(_ a ...) (syntax ((define a #f) ...))]))
```

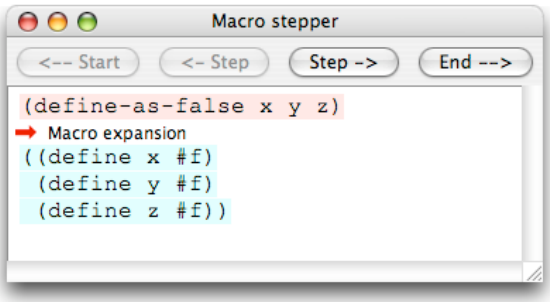
The macro definition is a function that consumes a syntax tree, named `stx`. The `syntax-case` construct de-structures the tree and binds pattern variables to its components. The `syntax` constructor produces a new syntax tree by replacing the pattern variables in its template with their values.

Using the macro, like thus:

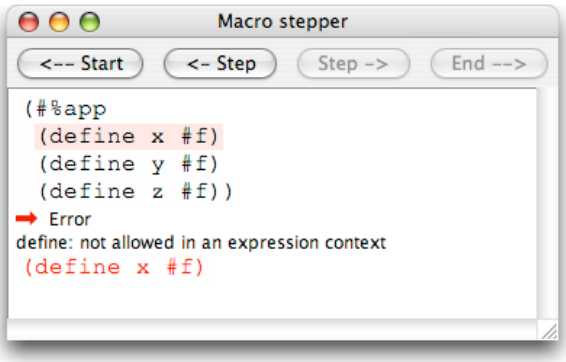
```
(def-false x y z)
```

immediately exposes a problem. The macro expander fails with an error explaining that definitions can't occur in an expression context. Of course, the problem is that the macro produces a list of terms, which the macro expander interprets as an application and which, in turn, may not contain any definitions.

Our macro stepper shows the sequence of macro expansion steps, one at a time:



Here we can see both the original macro form and the output of the macro application. The original appears at the top, the output of the first step at the bottom. The highlighted subterms on the top and bottom are the redex and contractum, respectively. The separator explains that this is a macro expansion step. At this point, an experienced Lisp or Scheme programmer recognizes the problem. A novice may need to see another step:



Here the macro expander has explicitly tagged the term as an application. The third step then shows the syntax error, highlighting the term *and* the context in which it occurred.

The macro debugger actually expands the entire term before it displays the individual steps. This allows programmers to skip to the very end of a macro expansion and to work backwards. The stepper supports this approach with a graphical user interface that permits programmers to go back and forth in an expansion and also to skip to the very end and the very beginning. The ideas for this interface have been borrowed from Clements's algebraic run-time stepper for PLT Scheme [3]; prior to that, similar ideas appeared in Lieberman's stepper [22] and Tolmach's SML debugger [29].

3.2 Syntax properties

Nearly all hygienic macro papers use the `or` macro to illustrate the problem of inadvertent variable capture:

```
(define-syntax (or stx)
  (syntax-case stx ()
    [(or e1 e2)
     (syntax (let ([tmp e1]) (if tmp tmp e2)))]))
```

In Scheme, the purpose of `(or a b)` is to evaluate `a` and to produce its value, unless it is false; if it is false, the form evaluates `b` and produces its value as the result.

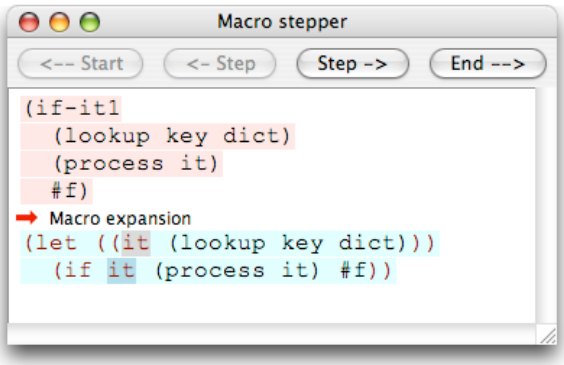
In order to keep `or` from evaluating its first argument more than once, the macro introduces a new variable for the first result. In Lisp-style macro expanders (or Scheme prior to 1986), the new `tmp` binding captures any free references to `tmp` in `e2`, thus interfering with the semantics of the macro and the program. Consequently, the macro breaks abstraction barriers. In Scheme, the new `tmp` identifier carries a mark or timestamp—introduced by the macro expander—that prevents it from binding anything but the two occurrences of `tmp` in the body of the macro-generated `let` [21]. This mark is vital to Scheme's macro expansion process, but no interface exists for inspecting the marks and the marking process directly.

Our macro debugger visually displays this scope information at every step. The display indicates with different text colors[3] from which macro expansion step every subterm originated. Furthermore, the programmer can select a particular subterm and see how the other subterms are related to it. Finally, the macro stepper can display a properties panel to show more detailed information such as identifier bindings and source locations.

The following example shows a programmer's attempt to create a macro called `if-it`, a variant of `if` that tries to bind the variable `it` to the result of the test expression for the two branches:

```
(define-syntax (if-it1 stx) ;; WARNING: INCORRECT
  (syntax-case stx ()
    [(if-it1 test then else)
     (syntax
      (let ([it test]) (if it then else)))]))
```

The same mechanism that prevents the inadvertent capture in the `or` example prevents the *intentional* capture here, too. With our macro debugger, the puzzled macro writer immediately recognizes why the macro doesn't work:



When the programmer selects an identifier, that identifier and all others with compatible binding properties are highlighted in the same color. Thus, in the screenshot above, the occurrence of `it` from the original program is *not* highlighted while the two macro-introduced occurrences are.

[3] Or numeric suffixes when there are no more easily distinguishable colors.

For completeness, here is the macro definition for a working version of `if-it`:

```
(define-syntax (if-it2 stx)
  (syntax-case stx ()
    [(if-it2 test then else)
     (with-syntax
         ([it (datum->syntax-object #'if-it2 'it)])
       (syntax
         (let ([it test])
           (if it then else))))]))
```

This macro creates an identifier named `it` with the lexical context of the original expression. The `syntax` form automatically unquotes `it` and injects the new identifier, with the correct properties, into the output. When the programmer examines the expansion of `(if-it2 a b c)`, all occurrences of `it` in `then` and `else` are highlighted now.

3.3 Scaling up

The preceding two subsections have demonstrated the workings of the macro debugger on self-contained examples. Some macros cannot be tested in a stand-alone mode, however, because it is difficult to extract them from the environment in which they occur.

One reason is that complex macros add entire sub-languages, not just individual features to the core. Such macros usually introduce local helper macros that are valid in a certain scope but nowhere else. For example, the `class` form in PLT Scheme, which is implemented as a macro, introduces a `super` form—also a macro—so that methods in derived classes can call methods in the base class. Since the definition of `super` depends on the rest of the class, it is difficult to create a small test case to explore its behavior. While restructuring such macros occasionally improves testability, *requiring restructuring for debugging is unreasonable*.

In general, the problem is that by the time the stepper reaches the term of interest, the context has been expanded to core syntax. Familiar landmarks may have been transformed beyond recognition. Naturally this prevents the programmer from understanding the macro as a linguistic abstraction in the original program. For the `class` example, when the expander is about to elaborate the body of a method, the `class` keyword is no longer visible; field and access control declarations have been compiled away; and the definition of the method no longer has its original shape. In such a situation, the programmer cannot see the forest for all the trees.

The macro debugger overcomes this problem with macro hiding. Specifically, the debugger implements a policy that determines which macros the debugger considers opaque; the programmer can modify this policy as needed. The macro debugger does not show the expansion of macros on this list, but it does display the expansions of the subtrees in the context of the original macro form. That is, the debugger presents steps that actually never happen and it presents terms that the expander actually never produces. Still, these intermediate terms are plausible and instructive, and for well-behaved macros,[4] they have the same meaning as the original and final programs.

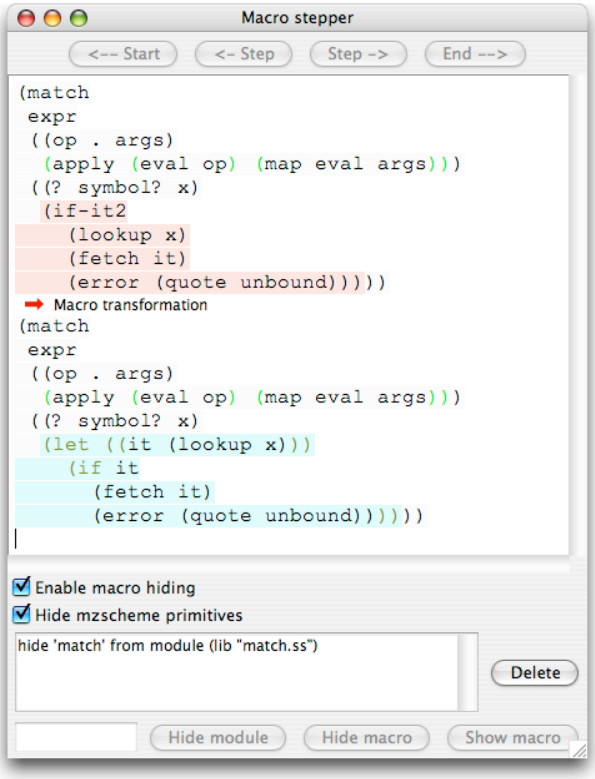
Consider the `if-it2` macro from the previous subsection. After testing the macro itself, the programmer wishes to employ it in the context of a larger program:

```
(match expr
  [(op . args)
   (apply (eval op) (map eval args))]
  [(? symbol? x)
   (if-it2 (lookup x)
           (fetch it)
           (error 'unbound))]))
```

This snippet uses the pattern-matching form called `match` from a standard (macro) library.

If the debugger had no "opaqueness policy" covering `match`, the macro expander and therefore the stepper would show the expansion of `if-it2` within the code produced by `match` macro. That code is of course a tangled web of nested conditionals, intermediate variable bindings, and failure continuations, all of which is irrelevant and distracting for the implementor of `if-it2`.

To eliminate the noise and focus on just the behavior of interest, the programmer instructs the macro debugger to consider `match` an opaque form. Then the macro debugger shows the expansion of the code above as a single step:



Although macro hiding and opaqueness policies simplify the story of expansion presented to the programmer, it turns out that implementing them is difficult and severely constrains the internal organization of the macro stepper. Before we can explain this, however, we must explain how Scheme expands macros.

4. Macro Expansion

Our model of macro expansion is an adaptation of Clinger and Rees's model [4], enriched with the lexical scoping mechanism of Dybvig et al. [6] and the phase separation of Flatt [11]. Figures 1 through 3 display the details.

Macro expansion is a recursive process that takes an expression and eliminates the macros, resulting in an expression in the core syntax of the language. The macro expander uses an environment to manage bindings and a phase number to manage staging. We use the following judgment to say the term $expr$ fully macro expands into $expr'$ in the syntactic environment E in phase number p:

$$p, E \vdash expr \Downarrow expr'$$

[4] For example, a macro that clones one of its subexpressions or inspects the structure of a subexpression is not well-behaved.

Terms	$expr$	::=	$identifier$
		|	$datum$
		|	$(expr\ \cdots\ expr)$
Identifiers	x, kw	::=	$symbol$
		|	$\mathrm{mark}(identifier, mark)$
		|	$\mathrm{subst}(ident, ident, symbol)$
Symbols	s	::=	countable set of names
Marks	$mark$	::=	countable set
Phases	p	::=	number: $0, 1, 2, \ldots$
Denotation	d	::=	`variable`
		|	$\langle$`primitive`, $symbol\rangle$
		|	$\langle$`macro`, $transformer\rangle$

Environments $E : symbol \times phase \rightarrow denotation$

Expansion relation	$p, E \vdash expr \Downarrow expr$
Macro step relation	$p, E \vdash expr \rightarrow expr$
Evaluation relation	$p, E \vdash expr \Downarrow_{eval} expr$

Figure 1. Semantic domains and relations

Figure 1 summarizes the domains and metavariables we use to describe the semantics.[5]

The structure of expressions varies from system to system. Lisp macro expanders operate on simple concrete syntax trees. Scheme macro systems are required to be hygienic—that is, they must respect lexical binding—and the expressions they manipulate are correspondingly complex. The hygiene principle [21, 4] of macro expansion places two requirements on macro expansion's interaction with lexical scoping:

1. Free identifiers introduced by a macro are bound by the binding occurrence apparent at the site of the macro's definition.
2. Binding occurrences of identifiers introduced by the macro (not taken from the macro's arguments) bind only other identifiers introduced by the same macro expansion step.

The gist of the hygiene requirement is that macros act "like closures" at compile time. Hence the meaning of a macro can be determined from its environment and its input; it does not depend on the context the macro is used in. Furthermore, if the macro creates a binding for a name that comes from the macro itself, it doesn't affect expressions that the macro receives as arguments.

Thinking of macro expansion in terms of substitution provides additional insight into the problem. There are two occurrences of substitution, and two kinds of capture to avoid. The first substitution consists of copying the macro body down to the use site; this substitution must not allow bindings in the context of the use site to capture names present in the macro body (hygiene condition 1). The second consists of substituting the macro's arguments into the body; names in the macro's arguments must avoid capture by bindings in the macro body (hygiene condition 2), even though the latter bindings are not immediately apparent in the macro's result.

Consider the following sample macro:

```
(define-syntax (munge stx)
  (syntax-case stx ()
    [(munge e)
     (syntax (mangle (x) e))]))
```

This macro puts its argument in the context of a use of a `mangle` macro. Without performing further expansion steps, the macro expander cannot tell if the occurrence of `x` is a binding occurrence. The expander must keep enough information to allow for both possibilities and delay its determination of the role of `x`.

The hygiene requirement influences the way lexical bindings are handled, and that in turn influences the structure of the expression representation. Technically, the semantics utilizes substitution and marking operations on expressions:

$$\mathrm{subst} : Expr \times Identifier \times Symbol \rightarrow Expr$$
$$\mathrm{mark} : Expr \times Mark \rightarrow Expr$$

Intuitively, these operations perform renaming and reversible stamping on all the identifiers contained in the given expression. These operations are generally done lazily for efficiency. There is an accompanying forcing operation called resolve

$$\mathrm{resolve} : Identifier \rightarrow Symbol$$

that sorts through the marks and substitutions to find the meaning of the identifier. Dybvig et al. [6] explain identifier resolution in detail and justify the lazy marking and renaming operations.

The expander uses these operations to implement variable renaming and generation of fresh names, but they don't carry the *meaning* of the identifiers; that resides in the expander's environment. This *syntactic environment* maps a symbol and a phase to a macro, name of a primitive form, or the designator `variable` for a value binding.

Determining the meaning of an identifier involves first resolving the substitutions to a symbol, and then consulting the environment for the meaning of the symbol in the current phase.

4.1 Primitive syntactic forms

Handling primitive syntactic forms generally involves recursively expanding the expression's subterms; sometimes the primitive applies renaming steps to the subterms before expanding them. Determining which rule applies to a given term involves resolving the leading keyword and consulting the environment. Consider the `lambda` rule from Figure 2. The keyword may be something other than the literal symbol `lambda`, but the rule applies to any form where the leading identifier has the *meaning* of the primitive `lambda` in the current environment.

The `lambda` syntactic form generates a new name for each of its formal parameters and creates a new body term with the old formals mapped to the new names—this is the renaming step. Then it extends the environment, mapping the new names to the `variable` designator, and expands the new body term in the extended environment. Finally, it re-assembles the `lambda` term with the new formals and expanded body.

When the macro expander encounters one of the `lambda`-bound variables in the body expression, it resolves the identifier to the fresh symbol from the renaming step, checks to make sure that the environment maps it to the `variable` designator (otherwise it is a misused macro or primitive name), and returns the resolved symbol.

The `if` and `app` (application) rules are simple; they just expand their subexpressions in the same environment.

The `let-syntax` form, which introduces local macro definitions, requires the most complex derivation rule (Figure 3). Like `lambda`, it constructs fresh names and applies a substitution to the body expression. However, it expands the right hand sides of the bindings using a phase number one greater than the current phase number. This prevents the macro transformers, which exist at compile time, from accessing *run-time* variables.[6] The macro transformers are then *evaluated* in the higher phase, and the envi-

[5] For simplicity, we do not model the store. Flatt [11] presents a detailed discussion of the interaction between phases, environments, and the store.

[6] Flatt [11] uses phases to guarantee separate compilation on the context of modules that import and export macros, but those issues are beyond the scope of this discussion.

ronment is extended in the *original* phase with the mapping of the macro names to the resulting values. The body of the `let-syntax` form is expanded in the extended environment, but with the original phase number.

For our purposes the run-time component of the semantics is irrelevant; we simply assume that the macro transformers act as functions that compute the representation of a new term when applied to a representation of the macro use. Thus, we leave the evaluation relation ($\Downarrow_{eval}$) unspecified.

4.2 Macros

Expanding a macro application involves performing the immediate macro transformation and then expanding the resulting term. The transformation step, represented by judgments of the form:

$$p, E \vdash expr \rightarrow expr$$

essentially consists of retrieving the appropriate macro transformer function from the environment and applying it to the macro use. We assume an invertible mapping from terms to data:

$$\begin{array}{rcl} \text{reflect} & : & Expr \rightarrow Datum \\ \text{reify} & : & Datum \rightarrow Expr \end{array}$$

$$\text{reify}(\text{reflect}(expr)) \quad = \quad expr$$

Like the evaluation relation ($\Downarrow_{eval}$), the details [6] of this operation are unimportant.

The interesting part of the macro rule is the marking and unmarking that supports the second hygiene condition. Identifiers introduced by the macro should not bind occurrences of free variables that come from the macro's arguments. The macro expander must somehow distinguish the two. The expander marks the macro arguments with a unique mark. When the macro transformer returns a new term, the parts originating from arguments still have a mark, and those that are newly introduced do not. Applying the same mark again cancels out the mark on old expressions and results in marks on only the introduced expressions.

When a marked identifier is used in a binding construct, the substitution only affects identifiers with the same name *and* the same mark. This satisfies the requirements of the second hygiene condition.

4.3 Syntax properties

The semantics shows how one kind of syntax property (scoping information) is manipulated and propagated by the macro expansion process.

The macro systems of real Scheme implementations define various other properties. For example, some put source location information in the syntax objects, and this information is preserved throughout macro expansion. Run time tools such as debuggers and profilers in these systems can then report facts about the execution of the program in terms of positions in the original code.

PLT Scheme allows macro writers to attach information keyed by arbitrary values to syntax. This mechanism has given rise to numerous lightweight protocols between macros, primitive syntax, and language tools.

5. Implementation

Programmers think of macros as rewriting specifications, where macro uses are replaced with their expansions. Therefore a macro debugger should show macro expansion as a sequence of rewriting steps. These steps are suggestive of a reduction semantics, but in fact we have not formulated a reduction semantics for macro expansion.[7] For a reduction semantics to be as faithful as the natural semantics we have presented, it would have to introduce administrative terms that obscure the user's program. We prefer to present an incomplete but understandable sequence of steps containing only terms from the user's program and those produced by macro expansion.

This section describes how we use the semantics in the implementation of the stepper, and the relationship between the semantics and the information displayed to the user.

5.1 Overview

The structure of a debugger is like the structure of a compiler. It has a front end that sits between the user and the debugger's internal representation of the program execution, a "middle end" or optimizer that performs translations on the internal representation, and a back end that connects the internal representation to the program execution.

While information flows from a compiler's front end to the back end, information in a debugger starts at the back end and flows through the front end to the user. The debugger's back end monitors the low-level execution of the program, and the front end displays an abstract view of the execution to the user. The debugger's middle end is responsible for finessing the abstract view, "optimizing" it for user comprehension.

Figure 4 displays the flow of information through our debugger. We have instrumented the PLT Scheme macro expander to emit information about the expansion process. The macro debugger receives this low-level information as a stream of events that carry data representing intermediate subterms and renamings. The macro debugger parses this low-level event stream into a structure representing the derivation in the natural semantics that corresponds to the execution of the macro expander. These derivations constitute the debugger's intermediate representation.

The debugger's middle end operates on the derivation generated by the back end, computing a new derivation tree with certain branches pruned away in accordance with the macro hiding policy. Finally, the front end traverses the optimized intermediate representation of expansion, turning the derivation structure into a sequence of rewriting steps, which the debugger displays in a graphical view. This view supports the standard stepping navigation controls. It also decorates the text of the program fragments with colors and mark-ups that convey additional information about the intermediate terms.

5.2 The Back End: Instrumentation

The macro expander of PLT Scheme is implemented as a collection of mutually recursive functions. Figure 5 presents a distilled version of the main function (in pseudocode).

The `expand-term` function checks the form of the given term. It distinguishes macro applications, primitive syntax, and variable references with a combination of pattern matching on the structure of the term and environment lookup of the leading keyword. Macro uses are handled by applying the corresponding transformer to a marked copy of the term, then unmarking and recurring on the result. Primitive forms are handled by calling the corresponding primitive expander function from the environment. The initial environment maps the name of each primitive (such as `lambda`) to its primitive expander (`expand-primitive-lambda`). When the primitive expander returns, expansion is complete for that term. The definitions of some of the primitive expander functions are

[7] Bove and Arbilla [2], followed by Gasbichler [16], have formulated reduction systems that present the macro expansion process as an ordered sequence of states. Each state is a term in an explicit substitution syntax (plus additional attributes). Unfortunately, these semantics are complex and unsuitable as simple specifications for a reduction system. In comparison, Clements [3] uses beta-value and delta-value rules as a complete specification and proves his stepper correct.

$$\textsc{Lambda}\ \frac{E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{lambda}\rangle \qquad \mathit{formals} \text{ is a list of distinct identifiers}}{\begin{array}{c}\mathit{formals}' = \text{freshnames}_E(\mathit{formals}) \qquad \mathit{body}' = \text{subst}(\mathit{body}, \mathit{formals}, \mathit{formals}') \\ E' = E[\langle \mathit{formals}', p\rangle \mapsto \texttt{variable}] \qquad p, E' \vdash \mathit{body}' \Downarrow \mathit{body}''\end{array}}{p, E \vdash (\mathit{kw}\ \ \mathit{formals}\ \ \mathit{body}) \Downarrow (\mathit{kw}\ \ \mathit{formals}'\ \ \mathit{body}'')}$$

$$\textsc{Variable}\ \frac{\text{resolve}(x) = x' \qquad E(x', p) = \texttt{variable}}{p, E \vdash x \Downarrow x'}$$

$$\textsc{If}\ \frac{E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{if}\rangle \qquad p, E \vdash \mathit{test} \Downarrow \mathit{test}' \qquad p, E \vdash \mathit{then} \Downarrow \mathit{then}' \qquad p, E \vdash \mathit{else} \Downarrow \mathit{else}'}{p, E \vdash (\mathit{kw}\ \ \mathit{test}\ \ \mathit{then}\ \ \mathit{else}) \Downarrow (\mathit{kw}\ \ \mathit{test}'\ \ \mathit{then}'\ \ \mathit{else}')}$$

$$\textsc{Application}\ \frac{E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{app}\rangle \qquad \forall i \le n : p, E \vdash \mathit{expr} \Downarrow \mathit{expr}'}{p, E \vdash (\mathit{kw}\ \ \mathit{expr}_0\ \cdots\ \mathit{expr}_n) \Downarrow (\mathit{kw}\ \ \mathit{expr'}_0\ \cdots\ \mathit{expr'}_n)}$$

Figure 2. Primitive syntactic forms

$$\textsc{Let-Syntax}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{let-syntax}\rangle \qquad \text{each } \mathit{var}_i \text{ is a distinct identifier} \\ \forall i \le n : \mathit{var'}_i = \text{freshname}_E(\mathit{var}_i) \\ \mathit{vars} = \{\mathit{var}_0, \ldots, \mathit{var}_n\} \qquad \mathit{vars}' = \{\mathit{var'}_0, \ldots, \mathit{var'}_n\} \qquad \mathit{body}' = \text{subst}(\mathit{body}, \mathit{vars}, \mathit{vars}') \\ \forall i \le n : p+1, E \vdash \mathit{rhs}_i \Downarrow \mathit{rhs}'_i \qquad \forall i \le n : p+1, E \vdash \mathit{rhs}'_i \Downarrow_{\mathit{eval}} \mathit{transformer}_i \\ E' = E[\{\langle \mathit{var}'_i, p\rangle \mapsto \mathit{transformer}_i\}] \qquad p, E' \vdash \mathit{body}' \Downarrow \mathit{body}''\end{array}}{p, E \vdash (\mathit{kw}\ \ ((\mathit{var}_0\ \ \mathit{rhs}_0)\ \cdots\ (\mathit{var}_n\ \ \mathit{rhs}_n))\ \ \mathit{body}) \Downarrow \mathit{body}''}$$

$$\textsc{Macro}\ \frac{p, E \vdash \mathit{expr} \to \mathit{expr}' \qquad p, E \vdash \mathit{expr}' \Downarrow \mathit{expr}''}{p, E \vdash \mathit{expr} \Downarrow \mathit{expr}''}$$

$$\textsc{Macro-Step}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{macro}, \mathit{transformer}\rangle \qquad \mathit{mark} = \text{freshmark}_E \\ \mathit{expr} = (\mathit{kw}\ \ \mathit{form}_1\ \cdots\ \mathit{form}_n) \qquad \mathit{stx} = \text{reflect}(\text{mark}(\mathit{expr}, \mathit{mark})) \\ p+1, E \vdash (\mathit{transformer}\ \ \mathit{stx}) \Downarrow_{\mathit{eval}} \mathit{stx}' \qquad \mathit{expr}' = \text{mark}(\text{reify}(\mathit{stx}'), \mathit{mark})\end{array}}{p, E \vdash (\mathit{kw}\ \ \mathit{form}_1\ \cdots\ \mathit{form}_n) \to \mathit{expr}'}$$

Figure 3. Macro definitions and uses

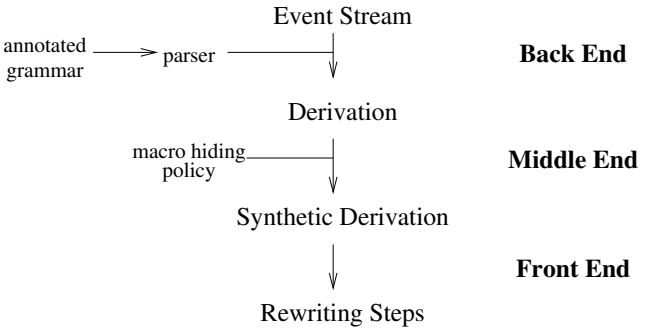


Figure 4. Implementation overview

given in Fig. 6; they recursively call `expand-term` on their subterms, as needed.

The shaded code in Fig. 5 and Fig. 6 represents our additions to the expander to emit debugging events. The calls to `emit-event` correspond to our instrumentation for the macro debugger. A call to `emit-event` send an event through a channel of communication to the macro debugger.

The events carry data about the state of the macro expander. Figure 7 shows a few of the event variants and the types of data they contain. A `visit` event indicates the beginning of an expansion step, and it contains the syntax being expanded. Likewise, the expansion of every term ends with a `return` event that carries the expanded syntax. The `enter-macro` and `exit-macro` events surround macro transformation steps, and the `macro-pre` and `macro-post` contain the marked versions of the starting and resulting terms. The `enter-primitive` and `exit-primitive` events surround all primitive form expansions. For every primitive, such as `if`, there is an event (`primitive-if`) that indicates that the macro expander is in the process of expanding that kind of primitive form. Primitives that create and apply renamings to terms send `rename` events containing the renamed syntax. The `next` signal separates the recursive expansion of subterms; `next-part` separates different kinds of subterms for primitives such as `let-syntax`.

5.3 The Back End: From Events to Derivations

The back end of the macro debugger transforms the low-level event stream from the instrumented expander into a derivation structure. This derivation structure corresponds to the natural semantics account of the expansion process.

The kinds of events in the stream determine the structure of the derivation, and the information carried by the events fills in the fields of the derivation objects. Figure 8 lists a few of the variants of the derivation datatype.

```
expand-term(term, env, phase) =
  emit-event("visit", term)
  case term of
    (kw . _)
      where lookup(resolve(kw), env, phase)
              = ("macro", transformer)
    => emit-event("enter-macro", term)
       let M = fresh-mark
       let term/M = mark(term, M)
       emit-event("macro-pre", term/M)
       let term2/M = transformer(term/M)
       emit-event("macro-post", term/M)
       let term2 = mark(term2/M, M)
       emit-event("exit-macro", term2)
       return expand-term(term2, env, phase)
    (kw . _)
      where lookup(resolve(kw), env, phase)
              = ("primitive", expander)
    => emit-event("enter-primitive", term)
       let term2 = expander(term, env, phase)
       emit-event("exit-primitive", term2)
       emit-event("return", term2)
       return term2
    id
      where lookup(resolve(id), env, phase)
              = "variable"
    => emit-event("enter-primitive", id)
       let term2 = expand-variable(id, env, phase)
       emit-event("exit-primitive", term2)
       emit-event("return", term2)
       return term2
    else
      => raise syntax error
```

Figure 5. Expansion function

Recall the inference rules from Fig. 2 and Fig. 3. The corresponding derivation structures contain essentially the same information in a different form. The first two fields of all derivation variants are the terms before and after expansion. The remaining fields are specific to the variant. In the `mrule` variant, the remaining field contains the derivation of the macro's result. In the `lambda` variant, the third field contains the new formal parameters and body expression after renaming, and the final field contains the derivation that represents the expansion of the renamed body expression. In the `if` variant, the three additional fields are the derivations for the three `if` subexpressions. The phase and environment parameters are not stored explicitly in the derivation structures, but they can be reconstructed for any subderivation from its context.

Creating structured data from unstructured sequences is a parsing problem. By inspecting the order of calls to `emit-event`, recursive calls to `expand-term`, and calls to other auxiliary functions, it is possible to specify a grammar that describes the language of event streams from the instrumented expander. Figure 9 shows such a grammar. By convention, non-terminal names start with upper-case letters and terminal names start with lower-case letters.

The `ExpandTerm` non-terminal describes the events generated by the `expand-term` function (Fig. 5) in expanding a term, whether it is the full program or a subterm. It has two variants: one for macros and one for primitive syntax. The `Primitive` non-terminal has a variant for each primitive syntactic form, and the productions

```
expand-prim-lambda(term, env, phase) =
  emit-event("primitive-lambda")
  case term of
    (kw formals body)
      where formals is a list of identifiers
      => let formals2 = freshnames(formals)
         let env2 =
           extend-env(env, formals, "variable", phase)
         let body2 = rename(body, formals, formals2)
         emit-event("rename", formals2, body2)
         let body3 = expand-term(body2, env2)
         return (kw formals2 body3)
    else => raise syntax error

expand-prim-if(term, env, phase) =
  emit-event("primitive-if")
  case term of
    (if test-term then-term else-term)
      => emit-event("next")
         let test-term2 = expand-term(test-term, env)
         emit-event("next")
         let then-term2 = expand-term(then-term, env)
         emit-event("next")
         let else-term2 = expand-term(else-term, env)
         return (kw test-term2 then-term2 else-term2)
    else => raise syntax error

expand-variable(id, env, phase) =
  let id2 = resolve(id)
  emit-event("variable", id2)
  return id2

expand-primitive-let-syntax(term, env, phase)
  emit-event("primitive-let-syntax", term)
  case term of
    (kw ([lhs rhs] ...) body)
      where each lhs is a distinct identifier
      => let lhss = (lhs ...)
         let rhss = (rhs ...)
         let lhss2 = freshnames(lhss)
         let body2 = rename(body, lhss, lhss2)
         emit-event("rename", lhss2 body2)
         let rhss2 =
           for each rhs in rhss:
             emit-event("next")
             expand-term(rhs, env, phase+1)
         let transformers =
           for each rhs2 in rhss2:
             eval(rhs2, env)
         emit-event("next-part")
         let env2 =
           extend-env(env, lhss2, transformers, phase)
         let body3 = expand-term(body2, env2, phase)
         return body3
    else => raise syntax error
```

Figure 6. Expansion functions for primitives and macros

`visit`	:	$Syntax$
`return`	:	$Syntax$
`enter-macro`	:	$Syntax$
`exit-macro`	:	$Syntax$
`macro-pre`	:	$Syntax$
`macro-post`	:	$Syntax$
`enter-primitive`	:	$Syntax$
`exit-primitive`	:	$Syntax$
`rename`	:	$Syntax \times Syntax$
`next`	:	()
`next-part`	:	()
`primitive-lambda`	:	()
`primitive-if`	:	()
`primitive-let-syntax`	:	()

Figure 7. Selected primitive events and the data they carry

for each primitive reflect the structure of the primitive expander functions from Fig. 6.

The macro debugger parses event streams into derivations according to this grammar. After all, a parse tree is the derivation proving that a sentence is in a language.[8] The action routines for the parser simply combine the data carried by the non-terminals—that is, the events—and the derivations constructed by the recursive occurrences of `ExpandTerm` into the appropriate derivation structures. There is one variant of the derivation datatype for each inference rule in the natural semantics (see Fig. 8).

5.4 Handling syntax errors

Both the derivation datatype from Fig. 8 and the grammar fragment in Fig. 9 describe only successful macro expansions, but a macro debugger must also deal with syntax errors. Handling such errors involves two additions to our framework:

1. new variants of the derivation datatype to represent interrupted expansions; after all, a natural semantics usually does not cope with errors
2. representation of syntax errors in the event stream, and additional grammar productions to recognize the new kinds of event streams

5.4.1 Derivations representing errors

A syntax error affects expansion in two ways:

1. The primitive expander function or macro transformer raises an error and aborts. We call this the *direct occurrence* of the error.
2. Every primitive expander in its context is interrupted, having completed some but not all of its work.

It is useful to represent these two cases differently. Figure 10 describes the extended derivation datatype.

For example, consider the expansion of this term:

```
(if x (lambda y) z)
```

The form of the `if` expression is correct; `expand-primitive-if` expands its subterms in order. When the expander encounters the `lambda` form, it calls the `expand-primitive-lambda` function, which rejects the form of the `lambda` expression and raises a syntax error. We represent the failed expansion of the `lambda` expression by wrapping a `prim:lambda` node in an `error-wrapper` node. The error wrapper also includes the syntax error raised.

That failure prevents `expand-primitive-if` from expanding the third subexpression and constructing a result term—but it did successfully complete the expansion of its first subterm. We represent the interrupted expansion of the `if` expression by wrapping the partially initialized `prim:if` node with an `interrupted-wrapper`. The interrupted wrapper also contains a tag that indicates that the underlying `prim:if` derivation has a complete derivation for its first subterm, it has an interrupted derivation for its second subterm, and it is missing the derivation for its third subterm.

5.4.2 The Error-handling Grammar

When an expansion fails, because either a primitive expander function or a macro transformer raises an exception, the macro expander places that exception at the end of the event stream as an `error` event. The event stream for the bad syntax example in Sec. 5.4.1 is:

```
visit enter-prim prim-if
  next visit enter-prim variable exit-prim return
  next visit enter-prim prim-lambda error
```

To recognize the event streams of failed expansions, we extend the grammar in the following way: for each non-terminal representing *successful* event streams, we add a new non-terminal that represents interrupted event streams. We call this the *interrupted* non-terminal, and by convention we name it by suffixing the original non-terminal name with "`/Error`." This interruption can take the form of an `error` event concerning the current rule or an interruption in the processing of a subterm.

Figure 11 shows two examples of these new productions.[9] The first variant of each production represents the case of a direct error, and the remaining variants represent the cases of errors in the expansions of subterms. The action routines for the first sort of error uses `error-wrapper`, and those for the second sort use `interrupted-wrapper`.

Finally, we change the start symbol to a new non-terminal called `Expansion` with two variants: a successful expansion `ExpandTerm` or an unsuccessful expansion `ExpandTerm/Error`.

The error-handling grammar is roughly twice the size of the original grammar. Furthermore, the new productions and action routines share a great deal of structure with the original productions and action routines. We therefore create this grammar automatically from annotations rather than manually adding the error-handling productions and action routines. The annotations specify positions for potential errors during the expansion process and potentially interrupted subexpansions. They come in two flavors: The first is the site of a potential error, written `(! tag)`, where tag is a symbol describing the error site. The second is a non-terminal that may be interrupted, written `(? NT tag)`, where `NT` is the non-terminal.

Figure 12 gives the definitions of the `PrimitiveLambda` and `PrimitiveIf` non-terminals from the annotated grammar. From these annotated definitions we produce the definitions of both the successful and interrupted non-terminals from Fig. 9 and Fig. 11, respectively.

The elaboration of the annotated grammar involves splitting every production alternate containing an error annotation into its successful and unsuccessful parts. This splitting captures the meaning of the error annotations:

- A potential error is either realized as an error that ends the event stream, or no error occurs and the event stream continues normally.

[8] Parser generators are widely available; we used the PLT Scheme parser generator macro [23].

[9] Figure 11 also shows that we rely on the delayed commitment to a particular production possible with LR parsers.

```
Derivation  ::=  (make-mrule Syntax Syntax Derivation)
             |   (make-prim:if Syntax Syntax Derivation Derivation Derivation)
             |   (make-prim:lambda Syntax Syntax Renaming Derivation)
             |   (make-prim:let-syntax Syntax Syntax Renaming DerivationList Derivation)
             |   ...
Renaming    ::=  Syntax × Syntax
```

Figure 8. Derivation structures

```
ExpandTerm ::= visit enter-macro macro-pre macro-post exit-macro ExpandTerm
             | visit enter-primitive Primitive exit-primitive return

Primitive ::= PrimitiveLambda
            | PrimitiveIf
            | PrimitiveApp
            | PrimitiveLetSyntax
            | ...

PrimitiveLambda ::= primitive-lambda rename ExpandTerm

PrimitiveIf ::= primitive-if next Expandterm next ExpandTerm next ExpandTerm

PrimitiveLetSyntax ::= primitive-let-syntax rename NextExpandTerms next-part ExpandTerm

NextExpandTerms ::= ε
                  | next ExpandTerm NextExpandTerms
```

Figure 9. Grammar of event streams

```
Derivation  ::=  ...
             |   (make-error-wrapper Symbol Exception Derivation)
             |   (make-interrupted-wrapper Symbol Derivation)
```

Figure 10. Extended derivation datatype

```
PrimitiveLambda/Error ::= primitive-lambda error
                        | primitive-lambda renames ExpandTerm/Error

PrimitiveIf/Error ::= primitive-if error
                    | primitive-if next ExpandTerm/Error
                    | primitive-if next ExpandTerm next ExpandTerm/Error
                    | primitive-if next ExpandTerm next ExpandTerm next ExpandTerm/Error
```

Figure 11. Grammar for interrupted primitives

```
PrimitiveLambda ::= primitive-lambda (! 'malformed) renames (? ExpandTerm)

PrimitiveIf ::= primitive-if (! 'malformed) next (? ExpandTerm 'test) next (? ExpandTerm 'then)
                next (? ExpandTerm 'else)
```

Figure 12. Grammar with error annotations

- A potentially interrupted subexpansion is either interrupted and ends the event stream, or it completes successfully and the event stream continues normally.

Naturally, we use a macro to elaborate the annotated grammar into the error-handling grammar. This is a nontrivial macro, and we made mistakes, but we were able to debug our mistakes using an earlier version of the macro stepper itself.

5.5 The Middle End: Hiding Macros

Once the back end has created a derivation structure, the macro debugger processes it with the user-specified macro hiding policy to get a new derivation structure. The user can change the policy many times during the debugging session. The macro debugger retains the original derivation, so updating the display involves only redoing the tree surgery and re-running the front end, which produces the rewriting sequence.

Conceptually, a macro hiding policy is simply a predicate on macro derivations. In practice, the user of the debugger controls the macro hiding policy by designating modules and particular macros as opaque or transparent. The debugger also provides named collections of modules, such as "mzscheme primitives," that can be hidden as a group. Policies may also contain more complicated provisions, and we are still exploring mechanisms for specifying these policies. We expect that time and user feedback will be necessary to find the best ways of building policies.

We refer to the original derivation as the true derivation, and the one produced by applying the macro policy as the synthetic derivation. The datatype of synthetic derivations contains an additional variant of node that does not correspond to any primitive syntactic form. This node contains a list of subterm expansions, where each subterm expansion consists of a derivation structure and a path representing the context of the subterm in the node's original syntax.

The middle end constructs the synthetic derivation by walking the true derivation and applying the policy predicate to every macro step. When the policy judges a macro step opaque, the debugger hides that step and the derivation for the macro's result, replacing the macro node with a synthetic node. The debugger then searches for *expanded subterms* within the hidden derivation. While searching through the subterms, it keeps track of the path through the opaque term to the subterms. When it finds an expanded subterm, it adds the derivation for the subterm's expansion, together with the path to that subterm, to the synthetic node.

Figure 13 illustrates one step of the macro hiding process. Suppose that the expansion of `let/cc` is marked as hidden. The debugger searches through the hidden derivation for subderivations corresponding to subterms of the opaque term. In the example from the figure, there is no subderivation for `k`, but there is a subderivation for $e1$. The macro hiding pass produces a new derivation with a *synthetic node* containing the derivation of $e1$ and the *path* to $e1$ in the original term. In this case, $e1$ can be reached in the term `(let/cc k` $e1$`)` through the path `(-- -- [])`.

If the expansion of $e1$ involves other opaque macros, then the debugger processes the derivation of $e1$ and places its corresponding synthetic derivation in the list of subterm derivations instead.

As a side benefit, macro hiding enables the debugger to detect a common beginner mistake: putting multiple copies of an input expression in the macro's output. If macro hiding produces a list of paths with duplicates (or more generally, with overlapping paths), the debugger reports an error to the programmer.

Engineering note 1: Macro hiding is complicated slightly by the presence of renaming steps. When searching for derivations for subterms, if the macro hider encounters a renaming step, it must also search for derivations for any subterms of the renamed term that correspond to subterms of the original term.

Engineering note 2: Performing macro hiding on the full language is additionally complicated by internal definition blocks. PLT Scheme partially expands the contents of a block to expose internal definitions, then transforms the block into a `letrec` expression and finishes handling the block by expanding the intermediate `letrec` expression.[10] Connecting the two passes of expansion for a particular term poses significant engineering problems to the construction of the debugger.

5.6 The Front End: Rewriting Steps

Programmers think of macro expansion as a term rewriting process, where macro uses are the redexes and primitive syntactic forms are the contexts. The front end of the debugger displays the process of macro expansion as a reduction sequence. More precisely, the debugger displays one rewriting step at a time, where each step consists of the term before the step and the term after the step, separated by an explanatory note.

The macro stepper produces the rewriting steps from the derivation produced by the middle end, which contains three sorts of derivation node. A macro step (`mrule`) node corresponds to a rewriting step, followed of course by the steps generated by the derivation for the macro's result. A primitive node generates rewriting steps for the renaming of bound variables, and it also generates rewriting steps from the expansion of its subterms. These rewriting steps occur in the context of the primitive form, with all of the previous subterms replaced with the results of their expansions.

For example, given subderivations `test`, `then`, and `else` for the three subterms of an `if` expression, we can generate the reduction sequence for the entire expression. We simply generate all the reductions for the first derivation and plug them into the original context. We build the next context by filling the first hole with the expanded version of the first subterm, and so on.

Opaque macros also act as expansion contexts. The synthetic nodes that represent opaque macros contain derivations paired with paths into the macro use's term. The paths provide the location of the holes for the contexts, and the debugger generates steps using the subderivations just as for primitive forms.

6. Conclusion

Despite the ever increasing complexity of Scheme's syntactic abstraction system, Scheme implementations have failed to provide adequate tools for stepping through macro expansions. Beyond the technical challenges that we have surmounted to implement our macro stepper, there are additional theoretical challenges in proving its correctness. Macro expansion is a complex process that thus far lacks a simple reduction semantics. We have therefore based our macro stepper on a natural semantics, with an ad hoc translation of derivations to reduction sequences.

First experiences with the alpha release of the debugger suggest that it is a highly useful tool, both for experts and novice macro developers. We intend to release the debugger to the wider Scheme community soon and expect to refine it based on the community's feedback.

Acknowledgments

We thank Matthew Flatt for his help in instrumenting the PLT Scheme macro expander.

References

[1] Eli Barzilay. Swindle. `http://www.barzilay.org/Swindle`.

[10] PLT Scheme macros are also allowed to partially expand their subterms and include the results in their output.

True derivation (before macro hiding):

$$\frac{p, E \vdash \texttt{(let/cc k } e1\texttt{)} \rightarrow \texttt{(call/cc (lambda (k) } e1\texttt{))} \qquad \dfrac{\dfrac{\dfrac{\Delta}{p, E' \vdash e1 \Downarrow e1'}}{\cdots}}{p, E \vdash \texttt{(call/cc (lambda (k) } e1\texttt{))} \Downarrow expr'}}{p, E \vdash \texttt{(let/cc k } e1\texttt{))} \Downarrow expr'}$$

Synthetic derivation (after macro hiding):

$$\frac{\left\langle \texttt{(-- -- [])}, \dfrac{\Delta}{p, E' \vdash e1 \Downarrow e1'} \right\rangle}{p, E \vdash \texttt{(let/cc k } e1\texttt{)} \Downarrow_h \texttt{(let/cc k } e1'\texttt{)}}$$

Figure 13. Macro hiding

[2] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *Proc. 1992 ACM Conference on LISP and Functional Programming*, pages 278–287, 1992.

[3] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. 10th European Symposium on Programming Languages and Systems*, pages 320–334, 2001.

[4] William Clinger and Jonathan Rees. Macros that work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[5] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proc. Fourth International Conference on Generative Programming and Component Engineering*, pages 373–388, 2005.

[6] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[7] Sebastian Egner. Eager comprehensions in scheme: The design of srfi-42. In *Proc. Sixth Workshop on Scheme and Functional Programming*, September 2005.

[8] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, 1985.

[9] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[10] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[11] Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[12] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[13] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[14] Daniel Friedman. Object-oriented style. In *International LISP Conference*, October 2003. Invited talk.

[15] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.

[16] Martin Gasbichler. *Fully-parameterized, First-class Modules with Hygienic Macros*. PhD thesis, Eberhard-Karls-Universität Tübingen, February 2006.

[17] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proc. Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 16–27, 2004.

[18] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[19] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[20] Oleg Kiselyov. A declarative applicative logic programming system, 2004–2006. `http://kanren.sourceforge.net`.

[21] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proc. 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[22] Henry Lieberman. Steps toward better debugging tools for lisp. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 247–255, 1984.

[23] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in scheme. In *Proc. Fifth Workshop on Scheme and Functional Programming*, pages 41–52, September 2004.

[24] PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2006-4-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[25] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proc. Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, 2004.

[26] Olin Shivers. The anatomy of a loop: a story of scope and control. In *Proc. Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.

[27] Dorai Sitaram. Programming in schelog. `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`.

[28] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, 1990.

[29] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[30] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, 1999.

[31] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[32] Andrew Wright and Bruce Duba. Pattern matching for scheme, 1995.

An Incremental Approach to Compiler Construction

Abdulaziz Ghuloum
Department of Computer Science, Indiana University, Bloomington, IN 47408
aghuloum@cs.indiana.edu

Abstract

Compilers are perceived to be magical artifacts, carefully crafted by the wizards, and unfathomable by the mere mortals. Books on compilers are better described as wizard-talk: written by and for a clique of all-knowing practitioners. Real-life compilers are too complex to serve as an educational tool. And the gap between real-life compilers and the educational toy compilers is too wide. The novice compiler writer stands puzzled facing an impenetrable barrier, "better write an interpreter instead."

The goal of this paper is to break that barrier. We show that building a compiler can be as easy as building an interpreter. The compiler we construct accepts a large subset of the Scheme programming language and produces assembly code for the Intel-x86 architecture, the dominant architecture of personal computing. The development of the compiler is broken into many small incremental steps. Every step yields a fully working compiler for a progressively expanding subset of Scheme. Every compiler step produces real assembly code that can be assembled then executed directly by the hardware. We assume that the reader is familiar with the basic computer architecture: its components and execution model. Detailed knowledge of the Intel-x86 architecture is not required.

The development of the compiler is described in detail in an extended tutorial. Supporting material for the tutorial such as an automated testing facility coupled with a comprehensive test suite are provided with the tutorial. It is our hope that current and future implementors of Scheme find in this paper the motivation for developing high-performance compilers and the means for achieving that goal.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; K.3.2 [*Computer and Information Science Education*]: Computer science education

Keywords Scheme, Compilers

1. Introduction

Compilers have traditionally been regarded as complex pieces of software. The perception of complexity stems mainly from traditional methods of teaching compilers as well as the lack of available examples of small and functional compilers for real languages.

Compiler books are polarized into two extremes. Some of them focus on "educational" toy compilers while the others focus on "industrial-strength" optimizing compilers. The toy compilers are too simplistic and do not prepare the novice compiler writer to construct a useful compiler. The source language of these compilers often lacks depth and the target machine is often fictitious. Niklaus Wirth states that "to keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice"[20]. On the other extreme, advanced books focus mainly on optimization techniques, and thus target people who are already well-versed in the topic. There is no gradual progress into the field of compiler writing.

The usual approach to introducing compilers is by describing the structure and organization of a finalized and polished compiler. The sequencing of the material as presented in these books mirrors the passes of the compilers. Many of the issues that a compiler writer has to be aware of are solved beforehand and only the final solution is presented. The reader is not engaged in the process of developing the compiler.

In these books, the sequential presentation of compiler implementation leads to loss of focus on the big picture. Too much focus is placed on the individual passes of the compiler; thus the reader is not actively aware of the relevance of a single pass to the other passes and where it fits in the whole picture. Andrew Appel states that "a student who implements all the phases described in Part I of the book will have a working compiler"[2]. Part I of Appel's book concludes with a 6-page chapter on "Putting it all together" after presenting 11 chapters on the different passes of Tiger.

Moreover, practical topics such as code generation for a real machine, interfacing to the operating system or to other languages, heap allocation and garbage collection, and the issues surrounding dynamic languages are either omitted completely or placed in an appendix. Muchnick states that "most of the compiler material in this book is devoted to languages that are well suited for compilation: languages that have static, compile-time type systems, that do not allow the user to incrementally change the code, and that typically make much heavier use of stack storage than heap storage"[13].

2. Preliminary Issues

To develop a compiler, there are a few decisions to be made. The source language, the implementation language, and the target architecture must be selected. The development time frame must be set. The development methodology and the final goal must be decided. For the purpose of our tutorial, we made the decisions presented below.

2.1 Our Target Audience

We do not assume that the reader knows anything about assembly language beyond the knowledge of the computer organization, memory, and data structures. The reader is assumed to have very limited or no experience in writing compilers. Some experience with writing simple interpreters is helpful, but not required.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

We assume that the reader has basic knowledge of `C` and the `C` standard library (e.g. `malloc`, `printf`, etc.). Although our compiler will produce assembly code, some functionality is easier to implement in `C`; implementing it directly as assembly routines distracts the reader from the more important tasks.

2.2 The Source Language

In our tutorial, we choose a subset of Scheme as the source programming language. The simple and uniform syntax of Scheme obviates the need for a lengthy discussion of scanners and parsers. The execution model of Scheme, with strict call-by-value evaluation, simplifies the implementation. Moreover, all of the Scheme primitives in the subset can be implemented in short sequences of assembly instructions. Although not all of Scheme is implemented in the first compiler, all the major compiler-related issues are tackled. The implementation is a middle-ground between a full Scheme compiler and a toy compiler.

In choosing a specific source language, we gain the advantage that the presentation is more concrete and eliminates the burden of making the connection from the abstract concepts to the actual language.

2.3 The Implementation Language

Scheme is chosen as the implementation language of the compiler. Scheme's data structures are simple and most Scheme programmers are familiar with the basic tasks such as constructing and processing lists and trees. The ability to manipulate Scheme programs as Scheme data structures greatly simplifies the first steps of constructing a compiler, since the issue of reading the input program is solved. Implementing a lexical-scanner and a parser are pushed to the end of the tutorial.

Choosing Scheme as the implementation language also eliminates the need for sophisticated and specialized tools. These tools add a considerable overhead to the initial learning process and distracts the reader from acquiring the essential concepts.

2.4 Choosing The Target Architecture

We choose the Intel-x86 architecture as our target platform. The x86 architecture is the dominant architecture on personal computers and thus is widely available.

Talking about compilers that are detached from a particular architecture puts the burden on the reader to make the connection from the abstract ideas to the concrete machine. Novice compiler writers are unlikely to be able to derive the connection on their own. Additionally, the compiler we develop is small enough to be easily portable to other architectures, and the majority of the compiler passes are platform independent.

2.5 Development Time Frame

The development of the compiler must proceed in small steps where every step can be implemented and tested in one sitting. Features that require many sittings to complete are broken down into smaller steps. The result of completing every step is a fully working compiler. The compiler writer, therefore, achieves progress in every step in the development. This is in contrast with the traditional development strategies that advocate developing the compiler as a series of passes only the last of which gives the sense of accomplishment. With our approach of incremental development, where every step results in a fully working compiler for some subset of Scheme, the risk of not "completing" the compiler is minimized. This approach is useful for people learning about compilers on their own, where the amount of time they can dedicate constantly changes. It is also useful in time-limited settings such as an academic semester.

2.6 Development Methodology

We advocate the following iterative development methodology:

1. Choose a small subset of the source language that we can compile directly to assembly.
2. Write as many test cases as necessary to cover the chosen subset of the language.
3. Write a compiler that accepts an expression (in the chosen subset of the source language) and outputs the equivalent sequence of assembly instructions.
4. Ensure that the compiler is functional, i.e. it passes all the tests that are written beforehand.
5. Refactor the compiler, if necessary, making sure that none of the tests are broken due to incorrect refactoring.
6. Enlarge the subset of the language in a very small step and repeat the cycle by writing more tests and extending the compiler to meet the newly-added requirements.

A fully working compiler for the given subset of the language is available at every step in the development cycle starting from the first day of development. The test cases are written to help ensure that the implementation meets the specifications and to guard against bugs that may be introduced during the refactoring steps. Knowledge of compilation techniques as well as the target machine is built incrementally. The initial overhead of learning the assembly instructions of the target machine is eliminated—instructions are introduced only when they are needed. The compiler starts small and is well focused on translating the source language to assembly, and every incremental step reinforces that focus.

2.7 Testing Infrastructure

The interface to the compiler is defined by one Scheme procedure, `compile-program`, that takes as input an s-expression representing a Scheme program. The output assembly is emitted using an `emit` form that routes the output of the compiler to an assembly file.

Defining the compiler as a Scheme procedure allows us to develop and debug the compiler interactively by inspecting the output assembly code. It also allows us to utilize an automated testing facility. There are two core components of the testing infrastructure: the test-cases and the test-driver.

The test cases are made of sample programs and their expected output. For example, the test cases for the primitive `+` may be defined as follows:

```
(test-section "Simple Addition")
(test-case '(+ 10 15) "25")
(test-case '(+ -10 15) "5")
...
```

The test-driver iterates over the test cases performing the following actions: (1) The input expression is passed to `compile-program` to produce assembly code. (2) The assembly code and a minimal run-time system (to support printing) are assembled and linked to form an executable. (3) The executable is run and the output is compared to the expected output string. An error is signaled if any of the previous steps fails.

2.8 The End Goal

For the purpose of this paper, we define the end goal to be writing a compiler powerful enough to compile an interactive evaluator. Building such a compiler forces us to solve many interesting problems.

A large subset of Scheme's core forms (`lambda`, `quote`, `set!`, etc) and extended forms (`cond`, `case`, `letrec`, internal `define` etc.) must be supported by the compiler. Although most of these forms are not essential, their presence allows us to write our programs in a more natural way. In implementing the extended forms, we show how a large number of syntactic forms can be added without changing the core language that the compiler supports.

A large collection of primitives (`cons`, `car`, `vector?`, etc.) and library procedures (`map`, `apply`, `list->vector`, etc.) need to be implemented. Some of these library procedures can be implemented directly, while others require some added support from the compiler. For example, some of the primitives cannot be implemented without supporting variable-arity procedures, and others require the presence of `apply`. Implementing a writer and a reader requires adding a way to communicate with an external run-time system.

3. Writing a Compiler in 24 Small Steps

Now that we described the development methodology, we turn our attention to the actual steps taken in constructing a compiler. This section is a brief description of 24 incremental stages: the first is a small language composed only of small integers, and the last covers most of the requirements of R^{5}RS. A more detailed presentation of these stages is in the accompanying extended tutorial.

3.1 Integers

The simplest language that we can compile and test is composed of the fixed-size integers, or fixnums. Let's write a small compiler that takes a fixnum as input and produces a program in assembly that returns that fixnum. Since we don't know yet how to do that, we ask for some help from another compiler that does know: gcc. Let's write a small `C` function that returns an integer:

```
int scheme_entry(){
  return 42;
}
```

Let's compile it using `gcc -O3 --omit-frame-pointer -S test.c` and see the output. The most relevant lines of the output file are the following:

```
1.          .text
2.          .p2align 4,,15
3.  .globl scheme_entry
4.          .type   scheme_entry, @function
5.  scheme_entry:
6.          movl    $42, %eax
7.          ret
```

Line 1 starts a text segment, where code is located. Line 2 aligns the beginning of the procedure at 4-byte boundaries (not important at this point). Line 3 informs the assembler that the `scheme_entry` label is global so that it becomes visible to the linker. Line 4 says that `scheme_entry` is a function. Line 5 denotes the start of the `scheme_entry` procedure. Line 6 sets the value of the `%eax` register to 42. Line 7 returns control to the caller, which expects the received value to be in the `%eax` register.

Generating this file from Scheme is straightforward. Our compiler takes an integer as input and prints the given assembly with the input substituted in for the value to be returned.

```
(define (compile-program x)
  (emit "movl $~a, %eax" x)
  (emit "ret"))
```

To test our implementation, we write a small `C` run-time system that calls our `scheme_entry` and prints the value it returns:

```
/* a simple driver for scheme_entry */
#include <stdio.h>
int main(int argc, char** argv){
  printf("%d\n", scheme_entry());
  return 0;
}
```

3.2 Immediate Constants

Values in Scheme are not limited to the fixnum integers. Booleans, characters, and the empty list form a collection of immediate values. Immediate values are those that can be stored directly in a machine word and therefore do not require additional storage. The types of the immediate objects in Scheme are disjoint, consequently, the implementation cannot use fixnums to denote booleans or characters. The types must also be available at run time to allow the driver to print the values appropriately and to allow us to provide the type predicates (discussed in the next step).

One way of encoding the type information is by dedicating some of the lower bits of the machine word for type information and using the rest of the machine word for storing the value. Every type of value is defined by a mask and a tag. The mask defines which bits of the integer are used for the type information and the tag defines the value of these bits.

For fixnums, the lower two bits ($mask = 11_b$) must be 0 ($tag = 00_b$). This leaves 30 bits to hold the value of a fixnum. Characters are tagged with 8 bits ($tag = 00001111_b$) leaving 24 bits for the value (7 of which are actually used to encode the ASCII characters). Booleans are given a 7-bit tag ($tag = 0011111_b$), and 1-bit value. The empty list is given the value 00101111_b.

We extend our compiler to handle the immediate types appropriately. The code generator must convert the different immediate values to the corresponding machine integer values.

```
(define (compile-program x)
  (define (immediate-rep x)
    (cond
      ((integer? x) (shift x fixnum-shift))
      ...))
  (emit "movl $~a, %eax" (immediate-rep x))
  (emit "ret"))
```

The driver must also be extended to handle the newly-added values. The following code illustrates the concept:

```
#include <stdio.h>
#define fixnum_mask       3
#define fixnum_tag        0
#define fixnum_shift      2
  ...
int main(int argc, char** argv){
  int val = scheme_entry();
  if((val & fixnum_mask) == fixnum_tag){
    printf("%d\n", val >> fixnum_shift);
  } else if(val == empty_list){
    printf("()\n");
  } ...
  return 0;
}
```

3.3 Unary Primitives

We extend the language now to include calls to primitives that accept one argument. We start with the simplest of these primitives: `add1` and `sub1`. To compile an expression in the form `(add1` e`)`, we first emit the code for e. That code would evaluate e placing its value in the `%eax` register. What remains to be done is incrementing

the value of the %eax register by 4 (the shifted value of 1). The machine instruction that performs addition/subtraction is addl/subl.

```
(define (emit-expr x)
  (cond
    ((immediate? x)
     (emit "movl $~a, %eax" (immediate-rep x)))
    ((primcall? x)
     (case (primcall-op x)
       ((add1)
        (emit-expr (primcall-operand1 x))
        (emit "addl $~a, %eax" (immediate-rep 1)))
       ...))
    (else ...)))
```

The primitives integer->char and char->integer can be added next. To convert an integer (assuming it's in the proper range) to a character, the integer (already shifted by 2 bits) is shifted further 6 bits to make up a total of *char-shift*, the result is then tagged with the *char-tag*. Converting a character to a fixnum requires a shift to the right by 6 bits. The choice of tags for the fixnums and characters is important for realizing this concise and potentially fast conversion.

We implement the predicates null?, zero?, and not next. There are many possible ways of implementing each of these predicates. The following sequence works for zero? (assuming the value of the operand is in %eax):

```
1.    cmpl   $0, %eax
2.    movl   $0, %eax
3.    sete   %al
4.    sall   $7, %eax
5.    orl    $63, %eax
```

Line 1 compares the value of %eax to 0. Line 2 zeros the value of %eax. Line 3 sets %al, the low byte of %eax, to 1 if the two compared values were equal, and to 0 otherwise. Lines 4 and 5 construct the appropriate boolean value from the one bit in %eax.

The predicates integer? and boolean? are handled similarly with the exception that the tag of the value must be extracted (using andl) before it is compared to the fixnum/boolean tag.

3.4 Binary Primitives

Calls to binary, and higher-arity, primitives cannot in general be evaluated using a single register since evaluating one subexpression may overwrite the value computed for the other subexpression. To implement binary primitives (such as +, *, char<?, etc.), we use a stack to save intermediate values of computations. For example, generating the code for (+ e_0 e_1) is achieved by (1) emitting the code for e_1, (2) emitting an instruction to save the value of %eax on the stack, (3) emitting the code for e_0, and (4) adding the value of %eax to the value saved on the stack.

The stack is arranged as a contiguous array of memory locations. A pointer to the base of the stack is in the %esp register. The base of the stack, 0(%esp), contains the return-point. The return-point is an address in memory where we return after computing the value and therefore should not be modified. We are free to use the memory locations above the return-point (-4(%esp), -8(%esp), -12(%esp), etc.) to hold our intermediate values.

In order to guarantee never overwriting any value that will be needed after the evaluation of an expression, we arrange the code generator to maintain the value of the stack index. The stack index is a negative number that points to the first stack location that is free. The value of the stack index is initialized to -4 and is decremented by 4 (the word-size, 4 bytes) every time a new value is saved on the stack. The following segment of code illustrates how the primitive + is implemented:

```
(define (emit-primitive-call x si)
  (case (primcall-op x)
    ((add1) ...)
    ((+)
     (emit-expr (primcall-operand2 x) si)
     (emit "movl %eax, ~a(%esp)" si)
     (emit-expr
       (primcall-operand1 x)
       (- si wordsize))
     (emit "addl ~a(%esp), %eax" si))
    ...))
```

The other primitives (-, *, =, <, char=?, etc.) can be easily implemented by what we know so far.

3.5 Local Variables

Now that we have a stack, implementing let and local variables is straightforward. All local variables will be saved on the stack and an environment mapping variables to stack locations is maintained. When the code generator encounters a let-expression, it first evaluates the right-hand-side expressions, one by one, saving the value of each in a specific stack location. Once all the right-hand-sides are evaluated, the environment is extended to associate the new variables with their locations, and code for the body of the let is generated in the new extended environment. When a reference to a variable is encountered, the code generator locates the variable in the environment, and emits a load from that location.

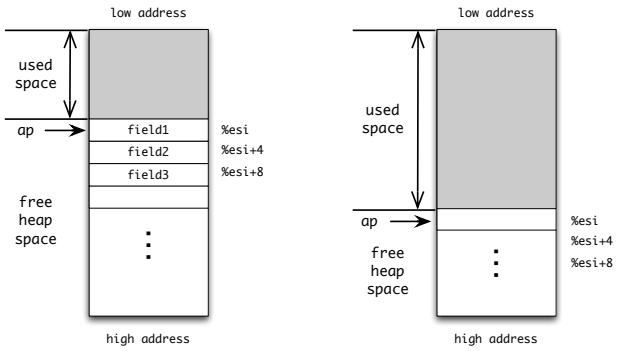
```
(define (emit-expr x si env)
  (cond
    ((immediate? x) ...)
    ((variable? x)
     (emit "movl ~a(%esp), %eax" (lookup x env)))
    ((let? x)
     (emit-let (bindings x) (body x) si env))
    ((primcall? x) ...)
    ...))

(define (emit-let bindings body si env)
  (let f ((b* bindings) (new-env env) (si si))
    (cond
      ((null? b*) (emit-expr body si new-env))
      (else
       (let ((b (car b*)))
         (emit-expr (rhs b) si env)
         (emit "movl %eax, ~a(%esp)" si)
         (f (cdr b*)
            (extend-env (lhs b) si new-env)
            (- si wordsize)))))))
```

3.6 Conditional Expressions

Conditional evaluation is simple at the assembly-level. The simplest implementation of (if test conseq altern) is:

```
(define (emit-if test conseq altern si env)
  (let ((L0 (unique-label)) (L1 (unique-label)))
    (emit-expr test si env)
    (emit-cmpl (immediate-rep #f) eax)
    (emit-je L0)
    (emit-expr conseq si env)
    (emit-jmp L1)
    (emit-label L0)
    (emit-expr altern si env)
    (emit-label L1)))
```

The code above first evaluates the test expression and compares the result to the false value. Control is transferred to the alternate



(A) Before allocation (B) After allocation

Figure 1. Illustration of the heap. The allocation pointer (ap) is is held in the `%esi` register and its value is always aligned on 8-byte boundaries. Individual objects are allocated at address `%esi` and the allocation pointer is bumped to the first boundary after the object.

code if the value of the test was false, otherwise, it falls through to the consequent.

3.7 Heap Allocation

Scheme's pairs, vector, strings, etc. do not fit in one machine word and must be allocated in memory. We allocate all objects from one contiguous area of memory. The heap is preallocated at the start of the program and its size is chosen to be large enough to accommodate our current needs. A pointer to the beginning of the heap is passed to `scheme_entry` to serve as the allocation pointer. We dedicate one register, `%esi`, to hold the allocation pointer. Every time an object is constructed, the value of `%esi` is incremented according to the size of the object.

The types of the objects must also be distinguishable from one another. We use a tagging scheme similar to the one used for fixnums, booleans, and characters. Every pointer to a heap-allocated object is tagged by a 3-bit tag (001_b for pairs, 010_b for vectors, 011_b for strings, 101_b for symbols, and 110_b for closures; 000_b, 100_b and 111_b were already used for fixnums and the other immediate objects). For this tagging scheme to work, we need to guarantee that the lowest three bits of every heap-allocated object is 000_b so that the tag and the value of the pointer do not interfere. This is achieved by always allocating objects at double-word (or 8-byte) boundaries.

Let's consider how pairs are implemented first. A pair requires two words of memory to hold its `car` and `cdr` fields. A call to `(cons 10 20)` can be translated to:

```
movl  $40,  0(%esi)        # set the car
movl  $80,  4(%esi)        # set the cdr
movl  %esi, %eax           # eax = esi | 1
orl   $1,   %eax
addl  $8,   %esi           # bump esi
```

The primitives `car` and `cdr` are simple; we only need to remember that the pointer to the pair is its address incremented by 1. Consequently, the `car` and `cdr` fields are located at -1 and 3 from the pointer. For example, the primitive `caddr` translates to:

```
movl  3(%eax),  %eax       # cdr
movl  3(%eax),  %eax       # cddr
movl  -1(%eax), %eax       # caddr
```

Vectors and strings are different from pairs in that they vary in length. This has two implications: (1) we must reserve one extra memory location in the vector/string to hold the length, and (2) after allocating the object, the allocation pointer must be aligned to the next double-word boundary (allocating pairs was fine because their size is a multiple of 8). For example, a call to the primitive `make-vector` translates to:

```
movl  %eax, 0(%esi)        # set the length
movl  %eax, %ebx           # save the length
movl  %esi, %eax           # eax = esi | 2
orl   $2,   %eax
addl  $11,  %ebx           # align size to next
andl  $-8,  %ebx           #    object boundary
addl  %ebx, %esi           # advance alloc ptr
```

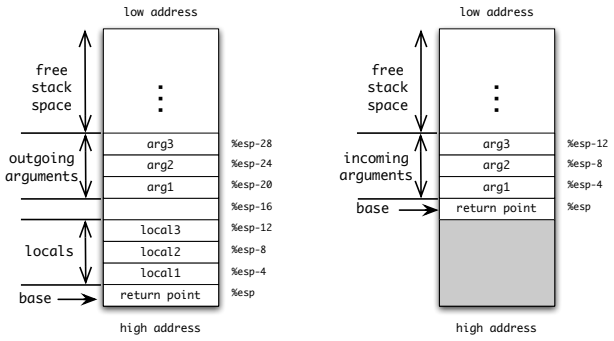
Strings are implemented similarly except that the size of a string is smaller than the size of a vector of the same length. The primitive `string-ref` (and `string-set!`) must also take care of converting a byte value to a character (and vise versa).

3.8 Procedure Calls

The implementation of procedures and procedure calls are perhaps the hardest aspect of constructing our compiler. The reason for its difficulty is that Scheme's `lambda` form performs more than one task and the compiler must tease these tasks apart. First, a `lambda` expression closes over the variables that occur free in its body so we must perform some analysis to determine the set of variables that are referenced, but not defined, in the body of a `lambda`. Second, `lambda` constructs a closure object that can be passed around. Third, the notion of procedure calls and parameter-passing must be introduced at the same point. We'll handle these issues one at a time starting with procedure calls and forgetting all about the other issues surrounding `lambda`.

We extend the language accepted by our code generator to contain top-level labels (each bound to a code expression containing a list of formal parameters and a body expression) and label calls.

```
<Prog>  ::= (labels ((lvar <LExpr>) ...) <Expr>)
<LExpr> ::= (code (var ...) <Expr>)
<Expr>  ::= immediate
          | var
          | (if <Expr> <Expr> <Expr>)
          | (let ((var <Expr>) ...) <Expr>)
          | (primcall prim-name <Expr> ...)
          | (labelcall lvar <Expr> ...)
```



(A) Caller's View (B) Callee's View

Figure 2. The view of the stack from (A) the Caller's side before making the procedure call, and (B) the Callee's side on entry to the procedure.

Code generation for the new forms is as follows:

- For the `labels` form, a new set of unique labels are created and the initial environment is constructed to map each of the `lvars` to its corresponding label.
- For each `code` expression, the label is first emitted, followed by the code of the body. The environment used for the body contains, in addition to the `lvars`, a mapping of each of the formal parameters to the first set of stack locations (-4, -8, etc.). The stack index used for evaluating the body starts above the last index used for the formals.
- For a `(labelcall lvar e ...)`, the arguments are evaluated and their values are saved in consecutive stack locations, skipping one location to be used for the return-point. Once all of the arguments are evaluated, the value of the stack-pointer, `%esp` is incremented to point to one word below the return-point. A `call` to the label associated with the `lvar` is issued. A `call` instruction decrements the value of `%esp` by 4 and saves the address of the next instruction in the appropriate return-point slot. Once the called procedure returns (with a value in `%eax`), the stack pointer is adjusted back to its initial position.

Figure 2 illustrates the view of the stack from the caller and callee perspective.

3.9 Closures

Implementing closures on top of what we have so far should be straightforward. First, we modify the language accepted by our code generator as follows:

- The form `(closure lvar var ...)` is added to the language. This form is responsible for constructing closures. The first cell of a closure contains the label of a procedure, and the remaining cells contain the values of the free variables.
- The `code` form is extended to contain a list of the free variables in addition to the existing formal parameters.
- The `labelcall` is replaced by a `funcall` form that takes an arbitrary expression as a first argument instead of an `lvar`.

The `closure` form is similar to a call to `vector`. The label associated with the `lvar` is stored at `0(%esi)` and the values of the variables are stored in the next locations. The value of `%esi` is tagged to get the value of the closure, and `%esi` is bumped by the required amount.

The `code` form, in addition to associating the formals with the corresponding stack locations, associates each of the free variables with their displacement form the closure pointer `%edi`.

The `funcall` evaluated all the arguments as before but skips not one but two stack locations: one to be used to save the current value of the closure pointer, and one for the return point. After the arguments are evaluated and saved, the operator is evaluated, and its value is moved to `%edi` (whose value must be saved to its stack location). The value of `%esp` is adjusted and an indirect call through the first cell of the closure pointer is issued. Upon return from the call, the value of `%esp` is adjusted back and the value of `%edi` is restored from the location at which it was saved.

One additional problem needs to be solved. The source language that our compiler accepts has a `lambda` form, and none of the `labels`, `code`, `closure` forms. So, Scheme input must be converted to this form before our code generator can accept it. The conversion is easy to do in two steps:

1. Free-variable analysis is performed. Every `lambda` expression appearing in the source program is annotated with the set of variables that are referenced but not defined in the body of the `lambda`. For example,

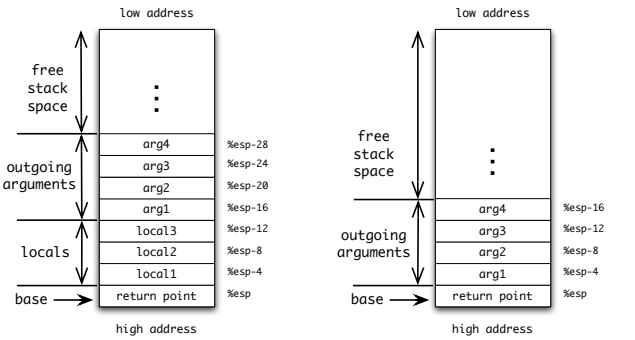


Figure 3. One way of implementing proper tail calls is by collapsing the tail frame. The figures show (A) the evaluation and placement of the arguments on the stack above the local variables, then (B) moving the arguments down to overwrite the current frame immediately before making the tail jump.

```
(let ((x 5))
  (lambda (y) (lambda () (+ x y))))
```

is transformed to:

```
(let ((x 5))
  (lambda (y) (x) (lambda () (x y) (+ x y))))
```

2. The `lambda` forms are transformed into `closure` forms and the codes are collected at the top. The previous example yields:

```
(labels ((f0 (code () (x y) (+ x y)))
         (f1 (code (y) (x) (closure f0 x y))))
  (let ((x 5)) (closure f1 x)))
```

3.10 Proper Tail Calls

The Scheme report requires that implementations be properly tail-recursive. By treating tail-calls properly, we guarantee that an unbounded number of tail calls can be performed in constant space.

So far, our compiler would compile tail-calls as regular calls followed by a return. A proper tail-call, on the other hand, must perform a `jmp` to the target of the call, using the same stack position of the caller itself.

A very simple way of implementing tail-calls is as follows (illustrated in Figure 3):

1. All the arguments are evaluated and saved on the stack in the same way arguments to nontail calls are evaluated.
2. The operator is evaluated and placed in the `%edi` register replacing the current closure pointer.
3. The arguments are copied from their current position of the stack to the positions adjacent to the return-point at the base of the stack.
4. An indirect `jmp`, not `call`, through the address in the closure pointer is issued.

This treatment of tail calls is the simplest way of achieving the objective of the requirement. Other methods for enhancing performance by minimizing the excessive copying are discussed later in Section 4.

3.11 Complex Constants

Scheme's constants are not limited to the immediate objects. Using the `quote` form, lists, vectors, and strings can be turned into constants as well. The formal semantics of Scheme require that quoted constants always evaluate to the same object. The following example must always evaluate to true:

```
(let ((f (lambda () (quote (1 . "H")))))
  (eq? (f) (f)))
```

So, in general, we *cannot* transform a quoted constant into an unquoted series of constructions as the following incorrect transformation demonstrates:

```
(let ((f (lambda () (cons 1 (string #\H)))))
  (eq? (f) (f)))
```

One way of implementing complex constants is by lifting their construction to the top of the program. The example program can be transformed to an equivalent program containing no complex constants as follows:

```
(let ((tmp0 (cons 1 (string #\H))))
  (let ((f (lambda () tmp0)))
    (eq? (f) (f))))
```

Performing this transformation before closure conversion makes the introduced temporaries occur as free variables in the enclosing `lambda`s. This increases the size of many closures, increasing heap consumption and slowing down the compiled programs.

Another approach for implementing complex constants is by introducing global memory locations to hold the values of these constants. Every complex constant is assigned a label, denoting its location. All the complex constants are initialized at the start of the program. Our running example would be transformed to:

```
(labels ((f0 (code () () (constant-ref t1)))
         (t1 (datum)))
  (constant-init t1 (cons 1 (string #\H)))
  (let ((f (closure f0)))
    (eq? (f) (f))))
```

The code generator should now be modified to handle the data labels as well as the two internal forms `constant-ref` and `constant-init`.

3.12 Assignment

Let's examine how our compiler treats variables. At the source level, variables are introduced either by `let` or by `lambda`. By the time we get to code generation, a third kind (free-variables) is there as well. When a `lambda` closes over a reference to a variable, we copied the *value* of the variable into a field in the closure. If more than one closure references the variable, each gets its own copy of the value. If the variable is assignable, then all references and assignments occurring in the code must reference/assign to the same location that holds the value of the the variable. Therefore, every assignable variable must be given one unique location to hold its value.

The way we treat assignment is by making the locations of assignable variables explicit. These locations cannot in general be stack-allocated due to the indefinite extent of Scheme's closures. So, for every assignable variable, we allocate space on the heap (a vector of size 1) to hold its value. An assignment to a variable x is rewritten as an assignment to the memory location holding x (via `vector-set!`) and references to x are rewritten as references to the location of x (via `vector-ref`).

The following example illustrates assignment conversion when applied to a program containing one assignable variable `c`:

```
(let ((f (lambda (c)
           (cons (lambda (v) (set! c v))
                 (lambda () c)))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
=>
(let ((f (lambda (t0)
           (let ((c (vector t0)))
             (cons (lambda (v) (vector-set! c 0 v))
                   (lambda () (vector-ref c 0)))))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
```

3.13 Extending the Syntax

With most of the core forms (`lambda`, `let`, `quote`, `if`, `set!`, constants, variables, procedure calls, and primitive calls) in place, we can turn to extending the syntax of the language. The input to our compiler is preprocessed by a pass, a macro-expander, which performs the following tasks:

- All the variables are renamed to new unique names through α-conversion. This serves two purposes. First, making all variables unique eliminates the ambiguity between variables. This makes the analysis passes required for closure and assignment conversion simpler. Second, there is no fear of confusing the core forms with procedure calls to local variables with the same name (e.g. an occurrence of `(lambda (x) x)` where `lambda` is a lexical variable).
- Additionally, this pass places explicit tags on all internal forms including function calls (`funcall`) and primitive calls (`primcall`).
- Extended forms are simplified to the code forms. The forms `let*`, `letrec`, `letrec*`, `cond`, `case`, `or`, `and`, `when`, `unless`, and internal `define` are rewritten in terms of the core forms.

3.14 Symbols, Libraries, and Separate Compilation

All of the primitives that we supported so far were simple enough to be implemented directly in the compiler as a sequence of assembly instructions. This is fine for the simple primitives, such as `pair?` and `vector-ref`, but it will not be practical for implementing more complex primitives such as `length`, `map`, `display`, etc..

Also, we restricted our language to allow primitives to occur only in the operator position: passing the value of the primitive `car` was not allowed because `car` has no value. One way of fixing this is by performing an inverse-η transformation:

$$\texttt{car} \Rightarrow \texttt{(lambda (x) (car x))}.$$

This approach has many disadvantages. First, the resulting assembly code is bloated with too many closures that were not present in the source program. Second, the primitives cannot be defined recursively or defined by using common helpers.

Another approach for making an extended library available is by wrapping the user code with a large `letrec` that defines all the primitive libraries. This approach is discouraged because the intermixing of user-code and library-code hinders our ability to debug our compiler.

A better approach is to define the libraries in separate files, compiling them independently, and linking them directly with the user code. The library primitives are initialized before control enters the user program. Every primitive is given a global location, or a label, to hold its value. We modify our compiler to handle two additional forms: `(primitive-ref` x`)` and `(primitive-set!` x v`)` which are analogous to `constant-ref` and `constant-init` that

we introduced in 3.11. The only difference is that global labels are used to hold the values of the primitives.

The first library file initializes one primitive: `string->symbol`. Our first implementation of `string->symbol` need not be efficient: a simple linked list of symbols suffices. The primitive `string->symbol`, as its name suggests, takes a string as input and returns a symbol. By adding the core primitives `make-symbol`[1] and `symbol-string`, the implementation of `string->symbol` simply traverses the list of symbols looking for one having the same string. A new symbol is constructed if one with the same name does not exist. This new symbol is then added to the list before it is returned.

Once `string->symbol` is implemented, adding symbols to our set of valid complex constants is straightforward by the following transformation:

```
  (labels ((f0 (code () () 'foo)))
    (let ((f (closure f0)))
      (eq? (funcall f) (funcall f))))
=>
  (labels ((f0 (code () () (constant-ref t1)))
           (t1 (datum)))
    (constant-init t1
      (funcall (primitive-ref string->symbol)
               (string #\f #\o #\o)))
    (let ((f (closure f0)))
      (eq? (funcall f) (funcall f))))
```

3.15 Foreign Functions

Our Scheme implementation cannot exist in isolation. It needs a way of interacting with the host operating system in order to perform Input/Output and many other useful operations. We now add a very simple way of calling to foreign `C` procedures.

We add one additional form to our compiler:

```
<Expr> ::= (foreign-call <string> <Expr> ...)
```

The `foreign-call` form takes a string literal as the first argument. The string denotes the name of the `C` procedure that we intend to call. Each of the expressions are evaluated first and their values are passed as arguments to the `C` procedure. The calling convention for `C` differs from the calling convention that we have been using for Scheme in that the arguments are placed below the return point and in reverse order. Figure 4 illustrates the difference.

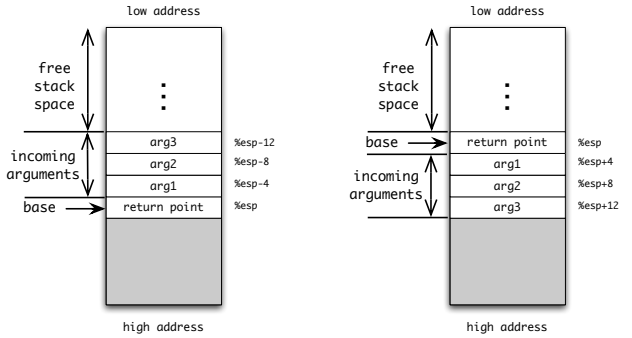
To accommodate the `C` calling conventions, we evaluate the arguments to a `foreign-call` in reverse order, saving the values on the stack, adjusting the value of `%esp`, issuing a `call` to the named procedure, then adjusting the stack pointer back to its initial position. We need not worry about the `C` procedure clobbering the values of the allocation and closure pointer because the Application Binary Interface (ABI) guarantees that the callee would preserve the values of the `%edi`, `%esi`, `%ebp` and `%esp` registers[14].

Since the values we pass to the foreign procedure are tagged, we would write wrapper procedures in our run-time file that take care of converting from Scheme to `C` values and back.

We first implement and test calling the `exit` procedure. Calling `(foreign-call "exit" 0)` should cause our program to exit without performing any output. We also implement a wrapper around `write` as follows:

```
ptr s_write(ptr fd, ptr str, ptr len){
  int bytes = write(unshift(fd),
                    string_data(str),
                    unshift(len));
  return shift(bytes);
}
```

[1] Symbols are similar to pairs in having two fields: a string and a value



(A) Parameter passing to Scheme (B) Parameter passing to `C`

Figure 4. The parameters to Scheme functions are placed on the stack above the return point while the parameters to `C` functions are placed below the return point.

3.16 Error Checking and Safe Primitives

Using our newly acquired ability to `write` and `exit`, we can define a simple `error` procedure that takes two arguments: a symbol (denoting the caller of `error`), and a string (describing the error). The `error` procedure would write an error message to the console, then causes the program to exit.

With `error`, we can secure some parts of our implementation to provide better debugging facilities. Better debugging allows us to progress with implementing the rest of the system more quickly since we won't have to hunt for the causes of segfaults.

There are three main causes of fatal errors:

1. Attempting to call non-procedures.
2. Passing an incorrect number of arguments to a procedure.
3. Calling primitives with invalid arguments. For example: performing `(car 5)` causes an immediate segfault. Worse, performing `vector-set!` with an index that's out of range causes other parts of the system to get corrupted, resulting in hard-to-debug errors.

Calling nonprocedures can be handled by performing a procedure check before making the procedure call. If the operator is not a procedure, control is transferred to an error handler label that sets up a call to a procedure that reports the error and exits.

Passing an incorrect number of arguments to a procedure can be handled by a collaboration from the caller and the callee. The caller, once it performs the procedure check, sets the value of the `%eax` register to be the number of arguments passed. The callee checks that the value of `%eax` is consistent with the number of arguments it expects. Invalid arguments cause a jump to a label that calls a procedure that reports the error and exits.

For primitive calls, we can modify the compiler to insert explicit checks at every primitive call. For example, `car` translates to:

```
   movl %eax, %ebx
   andl $7, %ebx
   cmpl $1, %ebx
   jne L_car_error
   movl -1(%eax), %eax
   ...
L_car_error:
   movl car_err_proc, %edi    # load handler
   movl $0, %eax              # set arg-count
   jmp *-3(%edi)              # call the handler
   ...
```

Another approach is to restrict the compiler to unsafe primitives. Calls to safe primitives are not open-coded by the compiler, instead, a procedure call to the safe primitive is issued. The safe primitives are defined to perform the error checks themselves. Although this strategy is less efficient than open-coding the safe primitives, the implementation is much simpler and less error-prone.

3.17 Variable-arity Procedures

Scheme procedures that accept a variable number of arguments are easy to implement in the architecture we defined so far. Suppose a procedure is defined to accept two or more arguments as in the following example:

```
(let ((f (lambda (a b . c) (vector a b c))))
  (f 1 2 3 4))
```

The call to `f` passes four arguments in the stack locations `%esp-4`, `%esp-8`, `%esp-12`, and `%esp-16` in addition to the number of arguments in `%eax`. Upon entry of `f`, and after performing the argument check, `f` enters a loop that constructs a list of the arguments last to front.

Implementing variable-arity procedures allows us to define many library procedures that accept any number of arguments including `+`, `-`, `*`, `=`, `<`, ..., `char=?`, `char<?`, ..., `string=?`, `string<?`, ..., `list`, `vector`, `string`, and `append`.

Other variations of `lambda` such as `case-lambda`, which allows us to dispatch different parts of the code depending on the number of actual arguments, can be implemented easily and efficiently by a series of comparisons and conditional jumps.

3.18 Apply

The implementation of the `apply` primitive is analogous to the implementation of variable-arity procedures. Procedures accepting variable number of arguments convert the extra arguments passed on the stack to a list. Calling `apply`, on the other hand, splices a list of arguments onto the stack.

When the code generator encounters an `apply` call, it generates the code in the same manner as if it were a regular procedure call. The operands are evaluated and saved in their appropriate stack locations as usual. The operator is evaluated and checked. In case of nontail calls, the current closure pointer is saved and the stack pointer is adjusted. In case of tail calls, the operands are moved to overwrite the current frame. The number of arguments is placed in `%eax` as usual. The only difference is that instead of calling the procedure directly, we `call`/`jmp` to the `L_apply` label which splices the last argument on the stack before transferring control to the destination procedure.

Implementing `apply` makes it possible to define the library procedures that take a function as well as an arbitrary number of arguments such as `map` and `for-each`.

3.19 Output Ports

The functionality provided by our compiler so far allows us to implement output ports easily in Scheme. We represent output ports by vector containing the following fields:

0. A unique identifier that allows us to distinguish output ports from ordinary vectors.
1. A string denoting the file name associated with the port.
2. A file-descriptor associated with the opened file.
3. A string that serves as an output buffer.
4. An index pointing to the next position in the buffer.
5. The size of the buffer.

The `current-output-port` is initialized at startup and its file descriptor is 1 on Unix systems. The buffers are chosen to be sufficiently large (4096 characters) in order to reduce the number of trips to the operating system. The procedure `write-char` writes to the buffer, increments the index, and if the index of the port reaches its size, the contents of the buffer are flushed using `s_write` (from 3.15) and the index is reset. The procedures `output-port?`, `open-output-file`, `close-output-port`, and `flush-output-port` are also implemented.

3.20 Write and Display

Once `write-char` is implemented, implementing the procedures `write` and `display` becomes straightforward by dispatching on the type of the argument. The two procedures are identical except for their treatment of strings and characters and therefore can be implemented in terms of one common procedure. In order to write the fixnums, the primitive `quotient` must be added to the compiler.

Implementing `write` in Scheme allows us to eliminate the now-redundant writer that we implemented as part of the `C` run-time system.

3.21 Input Ports

The representation of input ports is very similar to output ports. The only difference is that we add one extra field to support "unreading" a character which adds very minor overhead to the primitives `read-char` and `peek-char`, but greatly simplifies the implementation of the tokenizer (next step). The primitives added at this stage are `input-port?`, `open-input-file`, `read-char`, `unread-char`, `peek-char`, and `eof-object?` (by adding a special end-of-file object that is similar to the empty-list).

3.22 Tokenizer

In order to implement the `read` procedure, we first implement `read-token`. The procedure `read-token` takes an input port as an argument and using `read-char`, `peek-char`, and `unread-char`, it returns the next token. Reading a token involves writing an deterministic finite-state automata that mimics the syntax of Scheme. The return value of `read-token` is one of the following:

- A pair `(datum . `x`)` where x is a fixnum, boolean, character, string, or symbol that was encountered next while scanning the port.
- A pair `(macro . `x`)` where x denotes one of Scheme's predefined reader-macros: `quote`, `quasiquote`, `unquote`, or `unquote-splicing`.
- A symbol `left-paren`, `right-paren`, `vec-paren`, or `dot` denoting the corresponding non-datum token encountered.
- The end-of-file object if `read-char` returns the end-of-file object before we find any other tokens.

3.23 Reader

The `read` procedure is built as a recursive-descent parser on top of `read-token`. Because of the simplicity of the syntax (i.e. the only possible output is the eof-object, data, lists, and vectors) the entire implementation, including error checking, should not exceed 40 lines of direct Scheme code.

3.24 Interpreter

We have all the ingredients required for implementing an environment-passing interpreter for core Scheme. Moreover, we can lift the first pass of the compiler and make it the first pass to the interpreter as well. We might want to add some restriction to the language of the interpreter (i.e. disallowing `primitive-set!`) in order to prevent

the user code from interfering with the run-time system. We might also like to add different binding modes that determine whether references to primitive names refer to the actual primitives or to the current top-level bindings and whether assignment to primitive names are allowed or not.

4. Beyond the Basic Compiler

There are several axes along which one can enhance the basic compiler. The two main axes of enhancements are the feature-axis and the performance-axis.

4.1 Enhancing Features

The implementation presented in Section 3 featured many of the essential requirements for Scheme including proper tail calls, variable arity-procedures, and `apply`. It also featured a facility for performing foreign-calls that allows us to easily leverage the capabilities provided by the host operating system and its libraries. With separate compilation, we can implement an extended library of procedures including those required by the R^{5}RS or the various SRFIs. The missing features that can be added directly without changing the architecture of the compiler by much include:

- A full numeric tower can be added. The extended numerical primitives can either be coded directly in Scheme or provided by external libraries such as GNU MP.
- Multiple values are easy to implement efficiently using our stack-based implementation with very little impact on the performance of procedure calls that do not use multiple values [3].
- User-defined macros and a powerful module system can be added simply by compiling and loading the freely-available portable `syntax-case` implementation [7, 18].
- Our compiler does not handle heap overflows. Inserting overflow checks before allocation attempts should be simple and fast by comparing the value of the allocation pointer to an allocation limit pointer (held elsewhere) and jumping to an overflow handler label. A simple copying collector can be implemented first before attempting more ambitious collectors such as the ones used in Chez Scheme or The Glasgow Haskell Compiler [6, 12].
- Similarly, we did not handle stack overflows. A stack-based implementation can perform fast stack overflow checks by comparing the stack pointer to an end of stack pointer (held elsewhere) and then jumping to a stack-overflow handler. The handler can allocate a new stack segment and wire up the two stacks by utilizing an underflow handler. Implementing stack overflow and underflow handlers simplifies implementing efficient continuations capture and reinstatement [9].
- Alternatively, we can transform the input program into continuation passing style prior to performing closure conversion. This transformation eliminates most of the stack overflow checks and simplifies the implementation of `call/cc`. On the downside, more closures would be constructed at run-time causing excessive copying of variables and more frequent garbage collections. Shao et al. show how to optimize the representation of such closures [15].

4.2 Enhancing Performance

The implementation of Scheme as presented in Section 3 is simple and straightforward. We avoided almost all optimizations by performing only the essential analysis passes required for assignment and closure conversion. On the other hand, we have chosen a very compact and efficient representation for Scheme data structures. Such choice of representation makes error-checks faster and reduces the memory requirements and cache exhaustion.

Although we did not implement any source-level or backend optimizations, there is no reason why these optimization passes cannot be added in the future. We mention some "easy" steps that can be added to the compiler and are likely to yield high payoff:

- Our current treatment of `letrec` and `letrec*` is extremely inefficient. Better `letrec` treatment as described in [19] would allow us to (1) reduce the amount of heap allocation since most `letrec`-bound variables won't be assignable, (2) reduce the size of many closures by eliminating closures with no free-variables, (3) recognize calls to known procedures which allows us to perform calls to known assembly labels instead of making all calls indirect through the code pointers stored in closures, (4) eliminate the procedure check at calls to statically-known procedure, (5) recognize recursive calls which eliminates re-evaluating the value of the closures, (6) skip the argument-count check when the target of the call is known statically, and (7) `cons`ing up the rest arguments for known calls to procedures that accept a variable number of arguments.
- Our compiler introduces temporary stack locations for all complex operands. For example, `(+` e `4)` can be compiled by evaluating e first and adding 16 to the result. Instead, we transformed it to `(let ((`t_0 e`)) (+` t_0 `4))` which causes unnecessary saving and reloading of the value of e. Direct evaluation is likely to yield better performance unless good register allocation is performed.
- Our treatment of tail-calls can be improved greatly by recognizing cases where the arguments can be evaluated and stored in place. The greedy-shuffling algorithm is a simple strategy that eliminates most of the overhead that we currently introduce for tail-calls[4].
- None of the safe primitives were implemented in the compiler. Open-coding safe primitives reduces the number of procedure calls performed.
- Simple copy propagation of constants and immutable variables as well as constant-folding and strength-reduction would allow us to write simpler code without fear of inefficiencies. For example, with our current compiler, we might be discouraged from giving names to constants because these names would increase the size of any closure that contains a reference to them.

More sophisticated optimizations such as register allocation [5, 4, 16], inlining [17], elimination of run time type checks [10, 21], etc. could be targeted next once the simple optimizations are performed.

5. Conclusion

Compiler construction is not as complex as it is commonly perceived to be. In this paper, we showed that constructing a compiler for a large subset of Scheme that targets a real hardware is simple. The basic compiler is achieved by concentrating on the essential aspects of compilation and freeing the compiler from sophisticated analysis and optimization passes. This helps the novice compiler writers build the intuition for the inner-workings of compilers without being distracted by details. First-hand experience in implementing a basic compiler gives the implementor a better feel for the compiler's shortcomings and thus provide the motivation for enhancing it. Once the basic compiler is mastered, the novice implementor is better equipped for tackling more ambitious tasks.

Acknowledgments

I extend my thanks to Will Byrd, R. Kent Dybvig, and the anonymous reviewers for their insightful comments.

Supporting Material

The extended tutorial and the accompanying test suite are available for download from the author's website:

`http://www.cs.indiana.edu/~aghuloum`

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 1986.

[2] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, UK, 1998.

[3] ASHLEY, J. M., AND DYBVIG, R. K. An efficient implementation of multiple return values in scheme. In *LISP and Functional Programming* (1994), pp. 140–149.

[4] BURGER, R. G., WADDELL, O., AND DYBVIG, R. K. Register allocation using lazy saves, eager restores, and greedy shuffling. In *SIGPLAN Conference on Programming Language Design and Implementation* (1995), pp. 130–138.

[5] CHAITIN, G. J. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (New York, NY, USA, 1982), ACM Press, pp. 98–101.

[6] DYBVIG, R. K., EBY, D., AND BRUGGEMAN, C. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Tech. Rep. 400, Indiana University, 1994.

[7] DYBVIG, R. K., HIEB, R., AND BRUGGEMAN, C. Syntactic abstraction in scheme. *Lisp Symb. Comput. 5*, 4 (1992), 295–326.

[8] DYBVIG, R. K., HIEB, R., AND BUTLER, T. Destination-driven code generation. Tech. Rep. 302, Indiana University Computer Science Department, February 1990.

[9] HIEB, R., DYBVIG, R. K., AND BRUGGERMAN, C. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, NY, June 1990), vol. 25, pp. 66–77.

[10] JAGANNATHAN, S., AND WRIGHT, A. K. Effective flow analysis for avoiding runtime checks. In *2nd International Static Analysis Symposium* (September 1995), no. LNCS 983.

[11] KELSEY, R., CLINGER, W., AND (EDITORS), J. R. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (1998), 26–76.

[12] MARLOW, S., AND JONES, S. P. The new ghc/hugs runtime system.

[13] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[14] SCO. System V Application Binary Interface, Intel386™ Architecture Processor Supplement Fourth Edition, 1997.

[15] SHAO, Z., AND APPEL, A. W. Space-efficient closure representations. In *LISP and Functional Programming* (1994), pp. 150–161.

[16] TRAUB, O., HOLLOWAY, G. H., AND SMITH, M. D. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 142–151.

[17] WADDELL, O., AND DYBVIG, R. K. Fast and effective procedure inlining. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS '97)* (September 1997), vol. 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pp. 35–52.

[18] WADDELL, O., AND DYBVIG, R. K. Extending the scope of syntactic abstraction. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM Press, pp. 203–215.

[19] WADDELL, O., SARKAR, D., AND DYBVIG, R. K. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher Order Symbol. Comput. 18*, 3-4 (2005), 299–326.

[20] WIRTH, N. *Compiler Construction*. Addison Wesley Longman Limited, Essex, England, 1996.

[21] WRIGHT, A. K., AND CARTWRIGHT, R. A practical soft type system for scheme. *Transactions on Programming Languages and Systems* (1997).

SHard: a Scheme to Hardware Compiler

Xavier Saint-Mleux
Université de Montréal
saintmlx@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Jean-Pierre David
École Polytechnique de Montréal
jpdavid@polymtl.ca

Abstract

Implementing computations in hardware can offer better performance and power consumption than a software implementation, typically at a higher development cost. Current hardware/software co-design methodologies usually start from a pure software model that is incrementally transformed into hardware until the required performance is achieved. This is often a manual process which is tedious and which makes component transformation and reuse difficult. We describe a prototype compiler that compiles a functional subset of the Scheme language into synthesizable descriptions of dataflow parallel hardware. The compiler supports tail and non-tail function calls and higher-order functions. Our approach makes it possible for software developers to use a single programming language to implement algorithms as hardware components using standardized interfaces that reduce the need for expertise in digital circuits. Performance results of our system on a few test programs are given for FPGA hardware.

1. Introduction

Embedded systems combine software and hardware components. Hardware is used for interfacing with the real world and for accelerating the lower-level processing tasks. Software has traditionally been used for implementing the higher-level and more complex processing logic of the system and when future changes in functionality are expected. The partitioning of a system into its hardware and software components is a delicate task which must take into account conflicting goals including development cost, system cost, time-to-market, production volume, processing speed, power consumption, reliability and level of field upgradability.

Recent trends in technology are changing the trade-offs and making hardware components cheaper to create. Reconfigurable hardware is relatively recent and evolves rapidly, and can allow the use of custom circuits when field upgradability is desired or when production volume is expected to be low. Modern ASICs and FPGAs now contain enough gates to host complex embedded systems on a single chip, which may include tens of processors and dedicated hardware circuits. Power consumption becomes a major concern for portable devices. Specialized circuits, and in particular asynchronous and mixed synchronous/asynchronous circuits, offer better power usage than their equivalent software version running on a general purpose CPU or in synchronous logic [9][21][3].

The field of hardware/software co-design [11] has produced several tools and methodologies to assist the developer design and partition systems into hardware and software. Many tools present two languages to the developer, one for describing the hardware and the other for programming the software. This widespread approach has several problems. It requires that the developer learn two languages and processing models. The hardware/software interfaces may be complex, artificial and time consuming to develop. Any change to the partitioning involves re-writing substantial parts of the system. It is difficult to automate the partitioning process in this methodology.

Our position, which is shared by other projects such as SPARK-C [12], SpecC [8] and Handel-C [6], is that it is advantageous to employ a single language for designing the whole system (except perhaps the very lowest-level tasks). We believe that the partitioning of a system into hardware and software should be done by the compiler with the least amount of auxiliary partitioning information provided by the developer (e.g. command line options, pragmas, etc). This partitioning information allows the developer to optimize for speed, for space, for power usage, or other criteria. Moreover this information should be decoupled from the processing logic so that components can be reused in other contexts with different design constraints.

As a first step towards this long-term goal, we have implemented a compiler for a simple but complete parallel functional programming language which is fully synthesizable into hardware. Although our prototype compiler does not address the issue of automatic partitioning, it shows that it is possible to compile a general purpose programming language, and in particular function calls and higher-order functions, into parallel hardware.

We chose a subset of Scheme [17] as the source language for several reasons. Scheme's small core language allowed us to focus the development efforts on essential programming language features. This facilitated experimentation with various hardware specific extensions and allowed us to reuse some of the program transformations that were developed in the context of other Scheme compilers, in particular CPS conversion and 0-CFA analysis.

Our compiler, SHard, translates the source program into a graph of asynchronously connected instantiations of generic "black box" devices. Although the model could be fully synthesized with asynchronous components (provided a library with adequate models) we have validated our approach with an FPGA-based synchronous implementation where each asynchronous element is replaced by a synchronous Finite State Machine (FSM). Preliminary tests have been successfully performed using clockless implementations for some of the generic components, combined with synchronous FSMs for the others. Off-the-shelf synthesis tools are used to produce the circuit from the VHDL file generated by the compiler.

Throughout this project, emphasis has been put on the compilation process. To that effect, minimal effort has been put on opti-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

mization and on creating efficient implementations of the generic hardware components that the back-end instantiates. Demonstrating the feasibility of our compilation approach was the main concern. While SHard is a good proof of concept its absolute performance remains a future goal.

In Section 2 we give a general overview of our approach. In Section 3 the source language is described. The hardware building blocks of the dataflow architecture are described in Section 4 and Section 5 explains the compilation process. Section 6 explores the current memory management system and possible alternatives. Section 7 illustrates how behavioral simulations are performed and Section 8 outlines the current RTL implementation. Experimental results are given in Section 9. We conclude with related and future work in Section 10.

2. Overview

The implementation of functions is one of the main difficulties when compiling a general programming language to hardware. In typical software compilers a stack is used to implement function call linkage, but in hardware the stack memory can hinder parallel execution if it is centralized. The work on Actors [14] and the Rabbit Scheme compiler [13] have shown that a tail function call is equivalent to message passing. Tail function calls have a fairly direct translation into hardware. Non-tail function calls can be translated to tail function calls which pass an additional continuation parameter in the message. For this reason our compiler uses the Continuation Passing Style (CPS) conversion [2] to eliminate non-tail function calls. Each message packages the essential parts of a computational process and can be viewed as a process token moving through a dataflow architecture. This model is inherently parallel because more than one token can be flowing in the circuit. It is also energy efficient because a component consumes power only if it is processing a token. The main issues remaining are the representation of function closures in hardware and the implementation of message passing and tail calls in hardware.

Many early systems based on dataflow machines suffer from a memory bottleneck [10]. To reduce this problem our approach is to distribute throughout the circuit the memories which store the function closures. A small memory is associated with each function allocation site (`lambda`-expression) with free variables. The allocation of a cell in a closure memory is performed whenever the corresponding `lambda`-expression is evaluated. To avoid the need for a full garbage collector we deallocate a closure when it is called. Improvements on this simple but effective memory management model are proposed in Section 6.

By using a data flow analysis the compiler can tell which function call sites may refer to the closures contained in a specific closure memory. This is useful to minimize the number of busses and control signals between call sites and closure memories.

To give a feel for our approach we will briefly explain a small program. Figure 1 gives a program which sorts integers using the mergesort algorithm. The program declares an input channel (`cin`) on which groups of integers are received sequentially (as $\langle n, x_1, x_2, \ldots, x_n \rangle$), and an output channel (`cout`) on which the sorted integers are output. The program also declares functions to create pairs and lists as closures (`nil` and `cons`), the mergesort algorithm itself (functions `sort`, `split`, `merge`, and `revapp`), functions to read and write lists on the I/O channels (`get-list` and `put-list`) and a "main" function (`doio`) which reads a group, sorts it, outputs the result and starts over again. Note also that the predefined procedure `eq?` can test if two closures are the same (i.e. have the same address).

Figure 2 sketches the hardware components which are generated by the compiler to implement the `sort` function at line 33. The `sort` function can be called from three different places (at lines 57, 40 and 41; "callers" 1, 2 and 3 respectively) and "merge" components are used to route all function call requests to the function's body (A). The body starts with a fifo buffer (B) followed by the implementation of the test at line 34 ("stage" and "split" components, C). If the list is not empty, function `split` is called (line 36) after allocating a continuation closure for returning the result (D). This continuation, when called, allocates another continuation (the function at line 37) and calls `split`'s result (which is a function closure representing a pair) with it (E). Next, the test at line 38 is implemented like the previous one (F). If needed, two processes are forked with recursive calls to `sort` and their results are merged after both complete (G) (this is achieved with the `par` construct at line 40, which is syntactically like a `let` but evaluates all its binding expressions in parallel).

```
1. (letrec
2.     ((cin (input-chan cin))
3.      (cout (output-chan cout))
4.      (nil (lambda (_) 0))
5.      (cons (lambda (h t) (lambda (f) (f h t))))
6.      (revapp (lambda (L1 L2)
7.                (if (eq? nil L1)
8.                    L2
9.                    (L1 (lambda (h t)
10.                         (revapp t (cons h L2)))))))
11.     (split (lambda (L L1 L2)
12.              (if (eq? nil L)
13.                  (cons L1 L2)
14.                  (L (lambda (h t)
15.                       (split t (cons h L2) L1))))))
16.     (merge (lambda (L1 L2 L)
17.              (if (eq? nil L1)
18.                  (revapp L L2)
19.                  (if (eq? nil L2)
20.                      (revapp L L1)
21.                      (L1
22.                       (lambda (h1 t1)
23.                         (L2
24.                          (lambda (h2 t2)
25.                            (if (< h1 h2)
26.                                (merge t1
27.                                       (cons h2 t2)
28.                                       (cons h1 L))
29.                                (merge
30.                                 (cons h1 t1)
31.                                 t2
32.                                 (cons h2 L)))))))))))
33.     (sort (lambda (L)
34.             (if (eq? nil L)
35.                 nil
36.                 ((split L nil nil)
37.                  (lambda (L1 L2)
38.                    (if (eq? nil L2)
39.                        L1
40.                        (par ((s1 (sort L1))
41.                              (s2 (sort L2)))
42.                          (merge s1 s2 nil)))))))))
43.     (get-list (lambda (n)
44.                 (if (= 0 n)
45.                     nil
46.                     (cons (cin)
47.                           (get-list (- n 1))))))
48.     (put-list (lambda (L)
49.                 (if (eq? nil L)
50.                     (cout nil)
51.                     (L (lambda (h t)
52.                          (cout h)
53.                          (put-list t))))))
54.     (doio (lambda ()
55.             (let ((n (cin)))
56.               (let ((L (get-list n)))
57.                 (put-list (sort L))
58.                 (doio))))))
59.   (doio))
```

Figure 1. Mergesort program

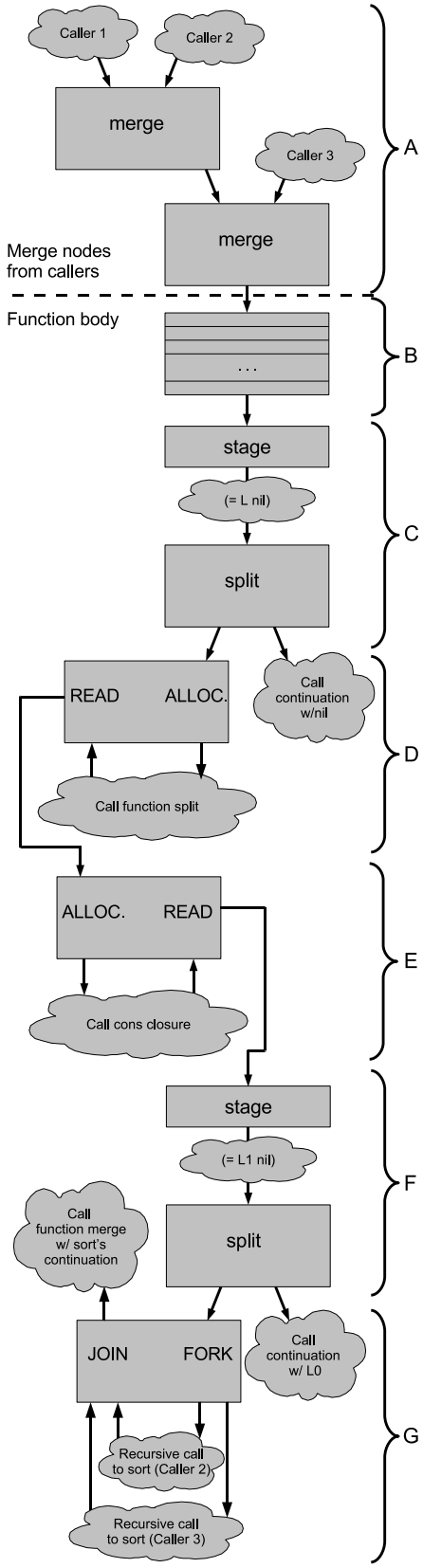


Figure 2. mergesort's `sort` function

3. Source Language

The source language is a lexically-scoped mostly functional language similar to Scheme [17]. It supports the following categories of expressions:

- Integer literal
- Variable reference
- Function creation: `(lambda (`*params*`)` *body*`)`
- Function call
- Conditional: `(if` *cond true-exp false-exp*`)`
- Binding: `let`, `letrec` and `par`
- Sequencing: `begin`
- I/O channel creation: `(input-chan` *name*`)` and `(output-chan` *name*`)`
- Global vectors: `make-vector`, `vector-set!` and `vector-ref`.

Primitive functions are also provided for integer arithmetic operations, comparisons on integers, bitwise operations and, for performance evaluation, timing. For example, the recursive factorial function can be defined and called as follows:

```
(letrec ((fact (lambda (n)
                 (if (< n 2)
                     1
                     (* n (fact (- n 1)))))))
  (fact 8))
```

In our prototype, the only types of data supported are fixed width integers, global vectors and function closures. Booleans are represented as integers, with zero meaning false and all other values meaning true. Closures can be used in a limited fashion to create data structures, as in the following example:

```
(let ((cons (lambda (h t)
              (lambda (f) (f h t))))
      (car (lambda (p)
             (p (lambda (h t) h)))))
  (let ((pair (cons 3 4)))
    (car pair)))
```

While this is a simple and effective way of supporting data structures, the programmer has to adapt to this model. The function call `(cons 3 4)` allocates memory for the closure containing the two integers, but this memory is reclaimed as soon as `pair` is called inside the `car` function; `pair` cannot be called again and the value `4` is lost. The only way to fetch the content of a closure is to call it and then recreate a similar copy using the data retrieved. Possible improvements are discussed in Section 6.

The `par` binding construct is syntactically and semantically similar to the `let` construct but it indicates that the binding expressions can be evaluated in parallel and that their evaluation must be finished before the body is evaluated. They can be seen as a calculation with a continuation that takes several return values. They can be used for manual parallelization when automatic parallelization (Section 5.1) is turned off or when expressions with side-effects may run concurrently.

The I/O channel creation forms create a functional representation of named input and output channels. Input channels are functions with no arguments that return the value read. Output channels take the value to be written as an argument and always return 0. The name given as an argument to `input-chan` and `output-chan` will be used as a signal name in the top-level VHDL circuit description. For example, the following specification creates a circuit that adds the values read from two different input channels, writes the sum on an output channel and starts over again:

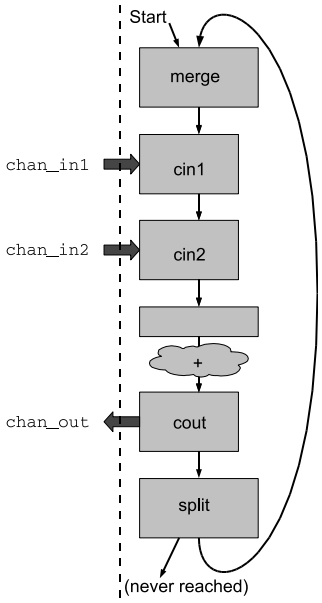


Figure 3. I/O example

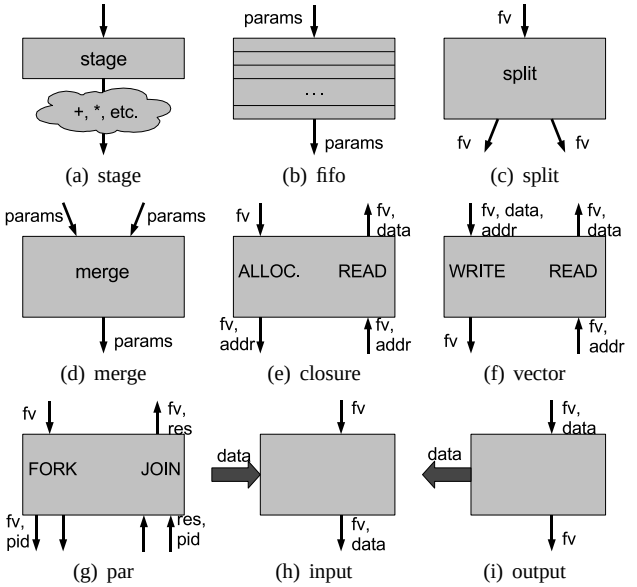


Figure 4. Generic Hardware Components

```
(let ((cin1 (input-chan chan_in1))
      (cin2 (input-chan chan_in2))
      (cout (output-chan chan_out)))
  (letrec ((doio (lambda ()
                   (cout (+ (cin1) (cin2)))
                   (doio))))
    (doio)))
```

This example is illustrated in Figure 3 using the components described in Section 4.

Like any other function with no free variables, channel procedures can be called any number of times since no closure allocation or deallocation is performed.

Mutable vectors are created with the `make-vector` primitive, which is syntactically similar to its Scheme equivalent. Currently, only statically allocated vectors are supported (i.e. vectors must be created at the top-level of the program). Our memory management model would have to be extended to support true Scheme vectors.

To simplify our explanations, we define two classes of expressions. *Trivial* expressions are literals, lambda expressions and references to variables. *Simple* expressions are either *trivial* expressions or calls of primitive functions whose arguments are *trivial* expressions.

4. Generic Hardware Components

The dataflow circuit generated by the compiler is a directed graph made of instantiations of the 9 generic components shown in Figure 4. The components are linked using unidirectional data busses which are the small arrows entering and leaving the components in the figure. Channels carry up to one message, or *token*, from the source component to the target component. Each channel contains a data bus and two wires for the synchronization protocol. The *request* wire, which carries a signal from the source to target, indicates the presence of a token on the bus. The *acknowledge* wire, which carries a signal from the target to the source, indicates that the token has been received at the target. The two signals implement a four-phase handshake protocol (i.e. ↑ Req, ↑ Ack, ↓ Req, ↓ Ack).

The following generic components are used in the system:

- **Stage** (Fig. 4(a)): Stages are used to bind new variables from *simple* expressions. Every `let` expression in which all bindings are from *simple* expressions are translated into stages in hardware; this is the case for all `let` expressions at the end of compilation. The stage component has an input channel that carries a token with all live variables in the expression that encloses the `let`. It has one output channel that sends a token with the same information, which then goes through a combinatorial circuit that implements all the *simple* expressions; the stage component is responsible for all synchronization so it must take into account the delay of the combinatorial part. The final token is sent to the component that implements the `let`'s body with all live variables at that point.
- **Fifo** (Fig. 4(b)): Fifos are used as buffers to accumulate tokens at the beginning of functions that might be called concurrently by several processes. Fifos are necessary in some situations to avoid deadlocks. They are conceptually like a series of n back-to-back stages but are implemented using RAM blocks in order to reduce the latency from input to output and the size of the circuit.
- **Split** (Fig. 4(c)): Split components are used to implement conditional (`if`) expressions. They have an input channel that receives a token with all live variables in the expression that encloses the conditional. The test expression itself is a reference to a boolean variable at the end of compilation so it is received directly as a wire carrying a 0 or a 1. Every token received is routed to the appropriate component through either one of two output channels, representing the *then* and *else* branches of the conditional expression. The appropriate branch will get a token carrying all live variables in the corresponding expression.
- **Merge** (Fig. 4(d)): Merge components are used to route tokens from a call site to the called function whenever there is more than one possible call site for a given function. Tokens received at the two input channels contain all the parameters of the function. In the case of a closure call, a pointer to the corresponding closure environment is also contained in the token. An arbiter ensures that every token received is sent to the component that implements the function's body, one at a time. Merge compo-

nents are connected together in a tree whenever there is more than two call sites for a function.

- **Closure** (Fig. 4(e)): Closure components are used to allocate and read the environments associated with function closures. They have two pairs of input and output channels, one for allocating and the other for reading. When a closure needs to be allocated, a token containing all live variables in the expression that encloses the closure declaration is received. All free variables in the function are saved at an unused address in a local RAM and a token containing that address, a tag identifying the function and all variables that are live in the declaration's continuation is sent to the component that implements that continuation. On a closure call, a token containing both the address of the closure's environment and the actual parameters is received, the free variables are fetched from the local RAM and a token containing the free variables and the actual parameters is sent to the component that implements the function's body. The closure component's read channel is connected to the output of the merge node(s) and the channel for the value read goes to the component that implements the function body. Since each closure has its dedicated block of RAM, the data width is exactly what is needed to save all free variables and no memory is wasted. Closure environments can be written or read in a single operation.
- **Vector** (Fig. 4(f)): Vector components are used to implement global vectors introduced through the `make-vector` primitive. They have two pairs of input and output channels, one for writing and another for reading. When the `vector-set!` primitive is called, a token is received with all live variables, an address and data to be written at that address. The data is written and a token with the live variables is sent as output. When the `vector-ref` primitive is called, a token containing all live variables and an address is received. Data is read from that address and sent in an output token along with the live variables. A block of RAM is associated with each vector and is sized accordingly.
- **Par** (Fig. 4(g)): Par components are used to implement `par` binding expressions. Like the closure component, the par component has an allocation and a reading part, respectively called *fork* and *join*. When a token is received for *fork*, it contains all the live variables of the `par` expression. All variables that are free in the `par`'s body are saved in a local RAM, much like for a closure environment; the corresponding address is an identifier for the `par` binding expressions' continuation. Then, tokens are sent simultaneously (forked) to the components that implement the binding expressions. Each of these parallel tokens contains the binding expression's continuation pointer and free variables. When a token is received for *join*, the binding expression's return value is saved in the local RAM along with the continuation's free variables. When the last token for a given identifier is received for *join* the return value is sent to the `par`'s body along with the other branches' return values and the free variables saved in the local RAM for that identifier, and the memory for that identifier is deallocated. Currently only two binding expressions are supported.
- **Input** (Fig. 4(h)): Input components implement all declared input channels in the circuit. It can be viewed like a simplified *join* part of a par component: it waits until it has received tokens from both inputs before sending one as output. One of the input tokens represents the control flow and contains all live variables in the call to the input function. The other input token contains the data present on the corresponding top-level signal of the circuit. The output token contains data from both input tokens.
- **Output** (Fig. 4(i)): Output components implement output channels. They act like a simplified *fork* part of a par component: whenever a token is received as input, two output tokens are sent simultaneously as output: one to the corresponding top-level signal of the circuit and one to the component that implements the continuation to the call to the output function.

The system has been designed so that all components can be implemented either as synchronous (clocked) or asynchronous components. For easy integration with the other synthesis and simulation tools available to us, our prototype currently uses clocked components reacting to the rising edge of the clock.

Input and output components can be implemented to support different kinds of synchronization. All experiments so far have been done using a four-phase handshake with passive inputs and active outputs: input components wait until they receive a request from the outside world and have to acknowledge it while output components send a request to the outside world and expect an acknowledgment. This allows linking of separately compiled circuits by simply connecting their IO channels together.

5. Compilation Process

The core of the compilation process is a pipeline of the phases described in this section. The 0-CFA is performed multiple times, as sub-phases of parallelization and inlining, and as a main phase by itself.

5.1 Parallelization

The compiler can be configured to automatically parallelize the computation. When this option is used, the compiler looks for sets of expressions which can be safely evaluated concurrently (side-effect free) and binds them to variables using a `par` construct. This is done only when at least two of the expressions are non-*simple*, since *simple* expressions are evaluated concurrently anyways (they are implemented as combinatorial circuits in a single stage) and the `par` construct produces a hardware component that implements the fork-join mechanism, which would be useless overhead in this case.

This stage is implemented as four different sub-stages. First a control flow analysis is performed on the program (see Section 5.5) in order to determine which expressions may actually have side-effects and which functions are recursive. Then, for all calls with arguments that are non-*simple*, those arguments are replaced with fresh variable references and the modified calls form the body of a `let` that binds those variables to the original arguments. For example,

```
(f (fact x) (- (fib y) 5) 3)
```

becomes

```
(let ((v_0 (fact x))
      (v_1 (- (fib y) 5)))
  (f v_0 v_1 3))
```

Next, all `let`s are analyzed and those for which all binding expressions have no side-effects and are non-*simple* are replaced by `par`s. Finally, the transformed program is analyzed to find all `par`s that may introduce an arbitrary number of tokens into the same part of the pipeline. These are the `par`s for which at least two binding expressions loop back to the `par` itself (e.g. a recursive function that calls itself twice). Any recursive function that can be called from the binding expressions is then tagged as "dangerous". The reason for this last step is that recursive functions are implemented as pipelines that feed themselves and each of these can only hold a given number of tokens at a given time before a deadlock occurs.

This tagging is used later in the compilation process to insert fifo buffers to reduce the possibility of deadlock.

5.2 CPS-Conversion

The compiler uses the CPS-Conversion to make the function call linkage of the program explicit by transforming all function calls into tail function calls. Functions no longer return a result; they simply pass the result along to another function using a tail call. Functions receive an extra parameter, the continuation, which is a function that represents the computation to be performed with the result. Where the original function would normally return a result to the caller, it now calls its continuation with this result as a parameter.

Since all functions now have an extra parameter, all calls must also be updated so that they pass a continuation as an argument. This continuation is made of the "context" of the call site embedded in a new lambda abstraction with a single parameter. The body of the continuation is the enclosing expression of the call where the call itself is replaced by a reference to the continuation's parameter. Syntactically the call now encloses in its new argument the expression that used to enclose it. For example,

```
(letrec ((fact (lambda (x)
                 (if (= 0 x)
                     1
                     (* x (fact (- x 1)))))))
  (+ (fact 3) 25))
```

becomes

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) (+ r 25)) 3))
```

There are two special cases. The program itself is an expression that returns a value so it should instead call a continuation with this result, but the normal conversion cannot be used in this case since there is no enclosing expression. The solution is to use a primitive called `halt` that represents program termination.

Because of the parallel fork-join mechanism, we also need to supply the parallel expressions with continuations. This is done in a similar way using a `pjoin` primitive which includes information about the `par` form that forked this process. This represents the fact that parallel sub-processes forked from a process must be matched with each other once they complete. For example,

```
(par ((x (f 3))
      (y (f 5)))
  ...)
```

becomes

```
(par pid_123
    ((x (f (lambda (r) (pjoin pid_123 r 0)) 3))
     (y (f (lambda (r) (pjoin pid_123 r 1)) 5)))
  ...)
```

`pid_123` is bound by the `par` at fork time and corresponds to the newly allocated address in the local RAM. `pjoin`'s last parameter (`0` or `1`) distinguishes the two sub-processes.

5.3 Lambda Lifting

Lambda lifting [16] is a transformation that makes the free variables of a function become explicit parameters of this function. Using this transformation, local functions can be lifted to the top-level of the program. Such functions have no free-variables and are called combinators. For example,

```
(let ((x 25))
  (let ((f (lambda (y) (+ x y))))
    (f 12)))
```

becomes

```
(let ((x 25))
  (let ((f (lambda (x2 y) (+ x2 y))))
    (f x 12)))
```

which is equivalent to

```
(let ((f (lambda (x2 y) (+ x2 y))))
  (let ((x 25))
    (f x 12)))
```

Since a combinator has no free-variables, it doesn't need to be aware of the environment in which it is called: all the values that it uses are explicitly passed as parameters. Combinators are closures that hold no data and therefore, in our system, we do not assign a closure memory to them. For example, if function `f` is not lambda lifted in the above example, it needs to remember the value of `x` between the function declaration and the function call; this would normally translate to a memory allocation at the declaration and a read at the call (see Section 5.6). After lambda lifting, `x` does not need to be known when `f` is declared since it will be explicitly passed as parameter `x2` on each call to `f`. The use of lambda lifting in our compiler helps to reduce the amount of memory used in the output circuit and to reduce latency.

Lambda lifting is not possible in all situations. For example:

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) r) 5))
```

This is a CPS-converted version of the classic factorial function. In this case, function `fact` needs to pass its result to continuation `k`, which can be the original continuation `(lambda (r) r)` or the continuation to a recursive call `(lambda (r) (k (* r x)))`. The continuation to a recursive call needs to remember about the parameters to the previous call to `fact` (`k` and `x`). We could add those free variables as parameters, like `(lambda (r k x) (k (* r x)))`, but then `fact` would need to know about the parameters to its previous call in order to be able to call its continuation, thus adding parameters to `fact` as well. Since `fact` is a recursive function and each recursive call needs to remember the parameters of the previous call, we would end up with a function that needs a different number of arguments depending on the context in which it is called, and this number could be arbitrarily large. Such cases are handled by closure conversion (Section 5.6) which identifies which closures actually contain data that needs to be allocated. In the `fact` example, the allocation of the free variables of the continuation to a recursive call (`k` and `x`) corresponds to the allocation of a stack frame in a software program.

5.4 Inlining

Inlining is a transformation which puts a copy of a function's body at the function's call site. In the circuit this corresponds to a duplication of hardware components. Although the resulting circuit is larger than could be, the circuit's parallelism is increased, which can yield faster computation. For this reason the compilation process includes an optional inlining phase.

The only information given to this phase by the developer is the maximum factor by which the code size should grow. Code size and circuit size is roughly approximated by the number of nodes in the corresponding AST. Since parallelism can only occur within

`par` expressions, inlining is done only in `par` binding expressions that have been tagged as "dangerous" by the parallelization phase (see Section 5.1). No inlining will occur in a program that does not exploit parallelism.

The inlining process is iterative and starts by inlining functions smaller than a given "inlining" size in all the identified expressions. If no function can be inlined and the desired growth factor has not been reached, this inlining size is increased and the process is iterated. Since inlining a function introduces new code in a `par`'s binding expressions, this can offer new candidate functions for inlining which may be much smaller than the current inlining size. The inlining size is therefore reset to its initial value after every iteration in which inlining actually occurred.

At each iteration, the number of callers for each inlinable function is calculated, functions are sorted from the most called to the least called and then treated in that order. The goal is to try to first duplicate components that have more chances of being a sequential bottleneck. 0-CFA is also done at each iteration in order to be aware of new call sites and the call sites that have vanished.

This method does not consider the fact that the size of the resulting circuit is not directly related to the size of the AST. In particular, the networking needed to connect calls to function bodies may grow quadratically as functions are duplicated. This is due to the fact that calls must be connected to every function possibly called, and the number of call sites also grows when code grows. An example of this is given in Section 9.

5.5 0-CFA

The 0-CFA (Control Flow Analysis [18]) performs a combined control flow and data flow analysis. Using the result of the 0-CFA, the compiler builds the control flow graph of the program. This is a graph that indicates which functions may be called at each function call site. This graph indicates how the circuit's components are interconnected, with each edge corresponding to a communication channel from the caller to the callee. When several edges point to the same node, we know that this function needs to be preceded by a tree of merge components (e.g. part A of Figure 2).

This analysis is also used for automatic parallelization and inlining as explained in Sections 5.1 and 5.4, and to assign locally unique identifiers to functions (see Section 5.7).

Abstract interpretation is used to gather the control flow information and that information is returned as an abstract value for each node in the AST. An abstract value is an upper bound of the set of all possible values that a given expression can evaluate to. In our case, all values other than functions are ignored and the abstract value is just a list of functions which represents the set containing those functions along with all non-function values.

5.6 Closure Conversion

Closure conversion is used to make explicit the fact that some functions are actually closures that contain data (free-variables); those are the functions that could not be made combinators by lambda lifting (Section 5.3). This conversion introduces two new primitives to the internal representation: `%closure` and `%clo-ref`. The `%closure` primitive is used to indicate that a function actually is a closure for which some data allocation must be made; its first parameter is the function itself and the rest are values to be saved in the closure memory. The `%clo-ref` is used within closures to indicate references to variables saved in the closure memory; it has two parameters: the first is a "self" parameter that indicates the address at which the data is saved and the second one is an offset within the data saved at that address (field number within a record), with `0` representing the function itself (not actually saved in memory). For example,

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) r) 5))
```

becomes

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     ((%clo-ref k 0) k 1)
                     (fact
                       (%closure
                         (lambda (self r)
                           ((%clo-ref
                              (%clo-ref self 2)
                              0)
                            (%clo-ref self 2)
                            (* r (%clo-ref self 1))))
                         x
                         k)
                       (- x 1)))))))
  (fact (%closure (lambda (self r) r)) 5))
```

5.7 Finalization

The finalization stage consists of three sub-stages: a trivial optimization, a "cosmetic" transformation to ease the job of the backend, and information gathering.

The first sub-stage merges sequences of embedded `let` expressions into a single `let`, when possible. It checks for a `let` in the body of another one and extracts the bindings in the embedded `let` that do not depend on variables declared by the embedding `let`. Those extracted bindings are moved up from the embedded `let` to the embedding one. If the embedded `let` ends up with an empty binding list, it is replaced by its own body as the body of the embedding `let`. For example,

```
(let ((a (+ x 7)))
  (let ((b (* y 6)))
    (let ((c (- a 3)))
      ...)))
```

becomes

```
(let ((a (+ x 7))
      (b (* y 6)))
  (let ((c (- a 3)))
    ...)))
```

This is done because each `let` translates directly to a pipeline stage in hardware; instead of operations being done in sequence in several stages, they are done concurrently in a single stage thus reducing latency and circuit size.

The next sub-stage is used to make explicit the fact that closures must be allocated before they are used. At this point in the compiler, arguments to calls are all values (literals or closures) or references to variables, so that a function call would be a simple connection between the caller and the callee. The only exception to this is that some closures contain data that must be allocated and these are represented by both a lambda identifier and an address that refers to the closure memory. To make everything uniform, closures that contain data are lifted in a newly created `let` that embeds the call. This way, we now have a special case of `let` that means "closure allocation" and the function call becomes a single stage where all arguments can be passed the same way. For example,

```
(foo 123 x (%closure (lambda (y) ...) a b))
```

becomes

```
(let ((clo_25 (%closure (lambda (y) ...) a b)))
  (foo 123 x clo_25))
```

so that it is clear that `a` and `b` are allocated in the closure memory in a stage prior to the function call.

The last sub-stage of finalization is used to assign locally unique identifiers to lambda abstractions to be used in the circuit instead of globally unique identifiers. The reason for this is that IDs take $\lceil \log_2 n \rceil$ bits to encode, where n can be the number of lambdas in the whole program (global IDs), or the number of lambdas in a specific subset (local IDs); our aim is to reduce the width of busses carrying those IDs. Since we have previously performed a 0-CFA, it is possible to know which lambdas may be called from a given call site. We first make the set of all those sets of functions and then merge the sets of functions that have elements in common until all are disjoint. This ensures that each lambda has an ID that, while not being necessarily unique in the program, gives enough information to distinguish this lambda from others in all call sites where it might be used.

5.8 Back-End

The back-end of the compiler translates the finalized Abstract Syntax Tree (AST) into a description of the circuit. The description is first output in an intermediate representation that describes the instantiation of several simple generic components and the data busses used as connections between them. This intermediate representation can be used for simulation and it is translated to VHDL through a secondary, almost trivial back-end (see Section 8).

Data busses are of two different types: `bus` and `join`:

- `(bus` *width val*`)`: describes a bus *width* bits wide with an initial value of *val*.
- `(join` $subbus_1$ `...` $subbus_n$`)`: describes a bus which is made of the concatenation of one or more other busses.

Integer literals and variable references are translated to busses; literals are constant busses while references are busses that get their value from the component that binds the variable (e.g. stage). All components that implement expressions through which some variable is live will have distinct input and output busses to carry its value, like if a new variable was bound at every pipeline stage.

As explained in Section 4, each `let` expression is translated to a stage component followed by a combinatorial circuit that implements the binding expressions and the component that implements the `let`'s body. The binding of a variable that represents a closure is translated to the *alloc* part of a closure component. Parallel bindings are translated to par components with the *fork* outputs and the *join* inputs connected to the binding expressions and the *join* output connected to the component that implements the `par`'s body. At this point in compilation, `letrec` bindings contain nothing else than function definitions so they are translated to the implementation of their body and all functions defined.

Function calls are translated to stage components where the combinatorial circuit is used to test the tag that identifies the function called and route the token accordingly. The result of the 0-CFA is used to determine which expressions call which functions. No stage is present if only one function can be called from a given point. Since all actual parameters are *trivial* expressions at this point in compilation, a connection from caller to callee is all that is needed.

Conditionals are translated to split components with each output connected to the component that implements the corresponding expression (*true-exp* or *false-exp*). As explained in Section 4, the condition is a *trivial* expression and its value is used directly to control a multiplexer.

Lambda abstraction declarations are translated to the bus that carries the function's or closure's identifying tag and address. Definitions are translated to the component that implements the function's body, possibly preceded by a tree of merge nodes and/or the *read* part of a closure component. The result of the 0-CFA is used to build the tree of merge nodes. Input and output channels are translated just like functions with an input or output component as a body.

Primitives may be translated in different ways: arithmetic primitives are translated to the equivalent combinatorial circuits while calls to the `timer` primitive – which returns the number of clock cycles since the last circuit reset – are similar to function calls to a global timer component. Other primitives are used internally by the compiler and each is translated in its own way. For example, the `halt` primitive terminates a process and the similar `pjoin` primitive indicates that a sub-process forked by a `par` has completed.

The compiler also adds an input and an output channel to the top-level circuit. The input channel carries tokens containing all free variables in the program and is used to start a new process; in most cases, this channel carries no data and is used only once since concurrent processes can be created using `par` expressions. The output channel carries tokens that contain the return value of the process; a token is output whenever the control reaches the `halt` primitive and indicates that a process has completed. Thus, the whole circuit can be seen as a function itself.

6. Memory Management

Memory management has been made as simple as possible since this is not our main research interest. As mentionned before, the memory associated to a closure is freed as soon as the closure is called. While this is enough to make our language Turing-complete, it imposes a very particular programming style, is error prone and makes inefficient tricks become indispensable in some situations.

The most desirable solution would, of course, be a full garbage collector as it normally exists in Scheme. Since closures can be stored in other closures (and this is always the case for continuations), a garbage collecting hardware component would need to be connected to all closure memories in a given system. The garbage collector would also need to be aware of all closures contained in tokens flowing in the system. Such a centralized component would hinder parallel execution and is in no way trivial to implement.

A reasonable alternative to a full garbage collector is to augment our memory management mechanism with a manual memory deallocation mechanism. This could be done as in many languages by using a "free" primitive. Closure memory components would need a new pair of input/output channels to support this, which would be connected to "free" call sites much like functions are connected to their call sites. It would also be possible to let the programmer indicate which closures are to be manually reclaimed and let the others be reclaimed automatically as is currently the case, thus reducing the size and latency of the resulting circuit.

Another issue is the amount of memory that is reserved for closures. Since each closure has its own block of RAM, this block has to be large enough to hold the largest number of closures that can exist concurrently, lest a deadlock might occur. Our prototype currently sets all closure memories to the same depth, which results in far more RAM being generated than necessary. One solution would be to use smaller closure memories and allow them to spill to a global memory; they would thus become local, distributed caches. Finding the optimal size for each local cache would be the main goal in order to minimize concurrent requests to the main memory. A non-trivial, multi-ported implementation of the global memory might be necessary in order to achieve good levels of parallelism.

Finally, the current implementation of vectors creates a bottleneck in parallel circuits since each vector is a single component and it cannot be duplicated like a function. A solution would be to split each vector into several independently accessible sub-vectors

controlled by a multiplexer which would route the request to the appropriate sub-vector.

7. Behavioral Simulation

The intermediate representation generated by the primary back-end is itself a Scheme program: it can be printed out in S-expression syntax and then executed to perform a simulation. This is done by using a simulator and behavioral descriptions of the components, both written in Scheme and included as external modules to the intermediate representation.

The simulator provides functions to manipulate wires and busses and to supply call-backs for some events: a signal transition, an absolute time from the beginning of simulation or a delay relative to the actual simulation time. In the behavioral simulator, it is therefore possible to write "when the input signal becomes 1, wait for 10ns and set the value of the output signal to 0" as:

```
(on input (lambda ()
            (if (= 1 (wire-value input))
                (lambda ()
                  (in 10 ;;time is in ns.
                      (lambda ()
                        (wire-update! output 1))))
                void)))
```

The program generated by the back-end also includes a test function which can be modified by hand to specify simulation parameters (input values, duration, etc). When the program is run, it produces output in VCD format (Value Change Dump, described in [1]). This output indicates the initial values of all signals in the circuit and all transitions that occurred during the simulation and can be sent to a standard waveform viewer (e.g. GTKWave).

8. Implementation

Hardware implementations are described using the VHDL language. All components listed in Section 4 are implemented as VHDL entities and architectures using generic parameters for bus widths, memory and fifo depths, etc. Most components have input signals for the global clock and reset signals.

For example, the stage VHDL component has an input channel and an output channel, and a `bus_width` generic parameter to specify the width of those channels. An internal register saves the input data at the rising edge of the clock on a successful handshake, and is cleared when the reset signal is asserted. Each channel is associated with a pair of wires that carry the request and acknowledge signals for synchronization; request signals go in the same direction as the data and acknowledge signals go the opposite way.

The top-level of the circuit is also translated from the Scheme program described above into a VHDL entity and architecture which instantiates all the necessary components. In addition to components described in Section 4, trivial combinatorial components like adders and equality testers are also used in the top-level.

The most difficult aspect of generating a VHDL circuit description is to handle `join` busses properly. There is no standard VHDL construct to express that some bus is in fact just an alias for the concatenation of other busses; these have to be translated to one-way assignments, either assigning the concatenation of several busses to a `join` bus or assigning a slice of a `join` to another bus. The rest is a straightforward translation of busses and components from the intermediate representation to VHDL, including bus renaming.

9. Results

We have tested our prototype on a number of programs to produce dataflow machines on an FPGA. The compiler's VHDL output

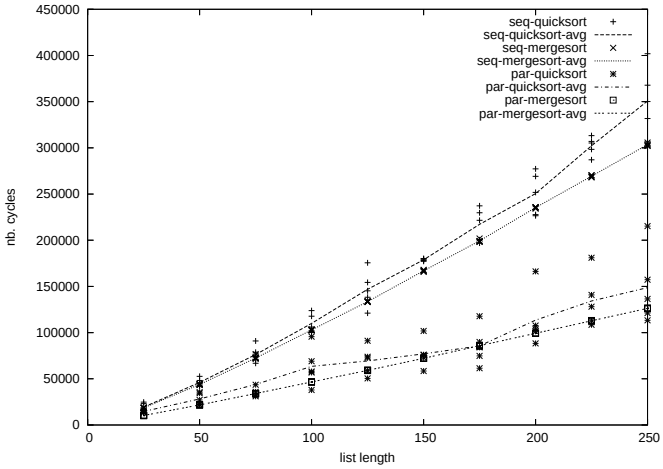


Figure 5. Sequential and parallel mergesort vs. quicksort, number of cycles as a function of list length

is fed to Altera's Quartus-II development environment. The only human intervention necessary at this point is the assignment of the circuit's external signals to FPGA pins; other constraints can also be given to the synthesis tool, for example to force it to try to produce a circuit that runs at a specific clock speed.

As an example, the quicksort and mergesort algorithms have been implemented in an Altera Stratix EP1S80 FPGA with a speed grade of -6. This FPGA contains 80,000 configurable cells. The list of elements is represented as a vector for quicksort and as a chain of closures for mergesort. The resulting circuits use about 11% and 14% of the reconfigurable logic and about 5% and 8% of the memory available in the FPGA, respectively, for lists of up to 256 16-bit integers and can run at clock rates above 80MHz. Also, mergesort is an algorithm for which the automatic parallelization stage of the compiler is useful.

Figure 5 shows the number of clock cycles required to sort lists of different lengths using mergesort and quicksort, for sequential and parallel versions of the algorithms. The parallel mergesort was automatically obtained by the compiler from a program without `par` expressions. Because of the vector mutations in the quicksort algorithm, the compiler could not obtain a parallel version automatically; it was necessary to manually insert a `par` expression for the recursive calls.

Figure 6 shows average clock cycles per element and compares sequential and parallel versions of both programs. It shows that a simple, sequential algorithm can gain a lot in terms of performance by using the parallelization stage of the compiler, or through simple modifications (changing `lets` to `pars`); performance is then limited by the amount of hardware used (e.g. components can be duplicated to gain more parallelism).

The fact that the quicksort algorithm is slower than the mergesort algorithm in our tests comes mainly from an inefficient implementation of vectors. Quicksort implemented using a chain of closures is, on average, faster than mergesort for sequential execution and about as fast for parallel execution.

Table 1 illustrates the effect of inlining (Section 5.4) on performance and circuit size. The program used for this test is the mergesort algorithm shown in Figure 1. In this program, the function which is inlined most often is `cons`, which has the effect of distributing the memory used to store the list in several independent memory blocks; with an inlining factor of 1.10, it is the only function that gets inlined and it is inlined five times out of a total

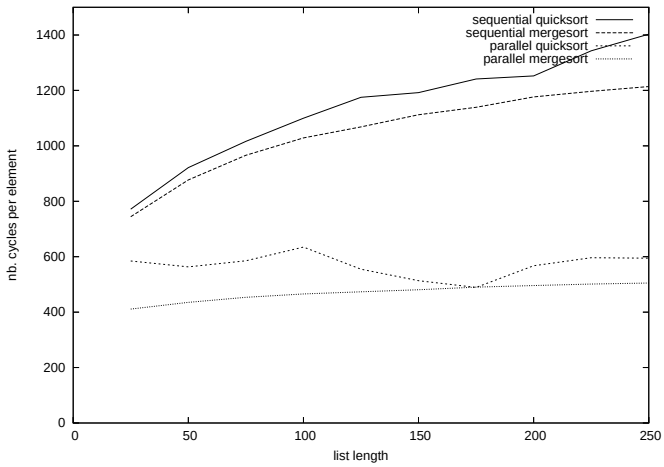


Figure 6. Parallel vs. Sequential mergesort and quicksort, average number of cycles per element as a function of list length

Inlining factor	% of logic	merge components	cycles to sort 250 elts.	% of baseline's (1.00) cycles
1.00	14	57	126,226	100.0
1.10	21	107	110,922	87.9
1.25	22	110	95,486	75.6
1.50	32	204	91,684	72.6
2.50	74	709	96,006	76.1

Table 1. Effect of inlining on mergesort

of seven call sites within `par`s. The proportion of logic is given for the Stratix EP1S80.

As mentioned in Section 5.4, the circuit size is not proportional to the AST size. To illustrate this, the number of merge components is given for each inlining factor. This outlines the fact that, by duplicating code, each function is potentially called from several more places. Area usage quickly becomes prohibitive as the inlining factor is increased. Also, more inlining does not always translate to a better performance: as the tree of merge components at each function entry gets bigger, the pipeline gets deeper and the latency increases; there is no need to have a lot more components than the maximum number of simultaneous tokens in the circuit.

To test the implementation of vectors we wrote a program which interprets a machine language for a custom 16-bit processor. Vectors are used to implement the RAM and the program memory. The instruction set contains 21 simple 16-bit instructions, some of which use a single immediate integer value. With the RAM and program memory both at 4096 elements deep, the circuit uses only 10% of the logic and 3% of the memory in a Stratix EP1S80. Unfortunately the execution speed is poor, in part because our language's lack of a `case` construct forced us to use nested `if`s to decode the instructions. It is exciting to consider that with some extensions to our system it might be possible to generate a "Scheme machine" processor by compiling an `eval` suitably modified for our system. Moreover, a multithreaded processor could be obtained easily by adding to the instruction set operations to fork new threads.

Tests have also been performed on the SHA-1 hashing algorithm. Since this algorithm always uses a fixed amount of memory, it has been written so that it does not use memory allocated data structures. Instead, each function receives all the values it needs as separate parameters. Input data is received in a stream from an input channel and new values are read only when the circuit is ready to process them. This has the effect of reducing latency since fewer closures have to be allocated, but it also means that tokens, and therefore data busses, can be very large. Closure memories for continuations also need to store more variables and the circuit ends up taking 39% of the logic and 23% of the memory in a Stratix EP1S80 device. This program highlights several situations in which simple optimizations could be added to the compiler to reduce the size of the circuit.

10. Conclusions

We have presented a compiler that automatically transforms a high level functional program into a parallel dataflow hardware description. The compilation process, from a Scheme-like language to VHDL, requires no user intervention and the approach has been validated on non-trivial algorithms. Our system handles tail and non-tail function calls, recursive functions and higher-order functions. This is done using closure memories which are distributed throughout the circuit, eliminating bottlenecks that could hinder parallel execution. The dataflow architecture generated is such that it could be implemented with power-efficient asynchronous circuits.

10.1 Related Work

Other research projects have studied the possibility of automatic synthesis of hardware architectures using software programming languages. Lava [4] allows the structural description of low-level combinatorial circuits in Haskell by the use of higher-order functions. It does not translate functional programs into hardware. Handel-C [6] is a subset of the C language which can be compiled directly to hardware, but it lacks support for features which are common in C, like pointers. Moreover it only supports inlined functions ("macros" which cannot be recursive). Scheme has also been applied to hardware synthesis in the context of the Scheme Machine project at Indiana University [20][15][5]. That work also does not support non-tail function calls and higher-order functions.

10.2 Future Work

In this work, our focus was to show that it is feasible to compile a functional description of a computation into a parallel circuit. We think it would be good to implement our generic hardware components in asynchronous logic to target very low power circuits. Asynchronous FPGAs [19] are being designed and these chips would be the perfect targets for our approach. As mentioned previously, an exciting prospect is the application of our compilation technique to hardware/software co-design for reconfigurable chips containing embedded processors and to Globally Asynchronous Locally Synchronous (GALS) architectures [7] which allow very high speed and massively parallel execution by eliminating the need for a global clock.

Several optimizations normally applied to software programs can be added to our compiler to produce more efficient circuits. For example, constant propagation can be used to reduce the width of busses and the size of memories, and even eliminate some superfluous closures. The simple inlining technique described in Section 5.4 could be replaced by a more clever one or one that can take into account the amount of logic available to implement the circuit or the desired level of parallelism. Common subexpression elimination, which compacts the circuit and reduces parallelism, may also be interesting to explore for space constrained applications.

As explained in Section 6, several improvements could be made to the memory management.

Our language lacks some useful constructs, such as `case` expressions, dynamically allocatable vectors, and data types, which would greatly enhance its expressiveness. A type system would

also be useful to determine the width of busses and memories and to perform static type checking.

References

[1] *IEEE Std 1364-2001 Verilog® Hardware Description Language*. IEEE, 2001.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302. ACM Press, 1989.

[3] G. M. Birtwistle and A. Davis, editors. *Asynchronous Digital Circuit Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, USA, 1998. ACM Press.

[5] R. G. Burger. The Scheme Machine. Technical Report Technical Report 413, Indiana University, Computer Science Department, August 1994.

[6] Celoxica. *Handel-C Language Reference Manual RM-1003-4.0*. http://www.celoxica.com, 2003.

[7] A. Chattopadhyay and Z. Zilic. GALDS: a complete framework for designing multiclock ASICs and SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(6):641–654, June 2005.

[8] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, editors. *SpecC: Specification Language and Methodology*. Springer, 2000.

[9] D. Geer. Is it time for clockless chip? *Computer*, pages 18–21, March 2005.

[10] C. Giraud-Carrier. A reconfigurable dataflow machine for implementing functional programming languages. *SIGPLAN Not.*, 29(9):22–28, 1994.

[11] R. Gupta and G. D. Micheli. Hardware/Software Co-Design. In *IEEE Proceedings*, volume 85, pages 349–365, March 1997.

[12] S. Gupta, N. Dutt, R. Gupta, and A. Nicola. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, New Delhi, India, January 2003.

[13] J. Guy L. Steele. Rabbit: A Compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.

[14] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[15] S. D. Johnson. Formal derivation of a scheme computer. Technical Report Technical Report 544, Indiana University Computer Science Department, September 2000.

[16] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, New York, NY, USA, 1985. Springer-Verlag Inc.

[17] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 Report on the Algorithmic Language Scheme. In *Higher-Order and Symbolic Computation*, volume 11, August 1998.

[18] O. G. Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Carnegie Mellon University, 1991.

[19] J. Teifel and R. Manohar. An Asynchronous Dataflow FPGA Architecture. *IEEE Transactions on Computers (special issue)*, November 2004.

[20] M. E. Tuna, S. D. Johnson, and R. G. Burger. Continuations in Hardware-Software Codesign. In *IEEE Proceedings of the International Conference on Computer Design*, pages 264–269, October 1994.

[21] C. Van Berkel, M. Josephs, and S. Nowick. Applications of asynchronous circuits. In *Proceedings of the IEEE*, volume 87, pages 223–233, Feb. 1999.

Automatic construction of parse trees for lexemes *

Danny Dubé
Université Laval
Quebec City, Canada
Danny.Dube@ift.ulaval.ca

Anass Kadiri
ÉPITA
Paris, France
Anass.Kadiri@gmail.com

Abstract

Recently, Dubé and Feeley presented a technique that makes lexical analyzers able to build parse trees for the lexemes that match regular expressions. While parse trees usually demonstrate how a word is generated by a context-free grammar, these parse trees demonstrate how a word is generated by a regular expression. This paper describes the adaptation and the implementation of that technique in a concrete lexical analyzer generator for Scheme. The adaptation of the technique includes extending it to the rich set of operators handled by the generator and reversing the direction of the parse trees construction so that it corresponds to the natural right-to-left construction of the lists in Scheme. The implementation of the adapted technique includes modifications to both the generation-time and the analysis-time parts of the generator. Uses of the new addition and empirical measurements of its cost are presented. Extensions and alternatives to the technique are considered.

Keywords Lexical analysis; Parse tree; Finite-state automaton; Lexical analyzer generator; Syntactic analysis; Compiler

1. Introduction

In the field of compilation, more precisely in the domain of syntactic analysis, we are used to associate the notion of parse tree, or derivation tree, to the notion of context-free grammars. Indeed, a parse tree can be seen as a demonstration that a word is generated by a grammar. It also constitutes a convenient structured representation for the word. For example, in the context of a compiler, the word is usually a program and the parse tree (or a reshaped one) is often the internal representation of the program. Since, in many applications, the word is quite long and the structure imposed by the grammar is non-trivial, it is natural to insist on building parse trees.

However, in the related field of lexical analysis, the notion of parse trees is virtually inexistent. Typically, the theoretical tools that tend to be used in lexical analysis are regular expressions and finite-state automata. Very often, the words that are manipulated are rather short and their structure, pretty simple. Consequently, the notion of parse trees is almost never associated to the notion of lexical analysis using regular expressions. However, we do not necessarily observe such simplicity in all applications. For instance, while numerical constants are generally considered to be simple lexical units, in a programming language such as Scheme [9], there are integers, rationals, reals, and complex constants, there are two notations for the complex numbers (rectangular and polar), there are different bases, and there are many kinds of prefixes and suffixes. While writing regular expressions for these numbers is manageable and matching sequences of characters with the regular expressions is straightforward, extracting and interpreting the interesting parts of a matched constant can be much more difficult and error-prone.

This observation has lead Dubé and Feeley [4] to propose a technique to build parse trees for lexemes when they match regular expressions. Until now, this technique had remained paper work only as there was no implementation of it. In this work, we describe the integration of the technique into a genuine lexical analyzer generator, SILex [3], which is similar to the Lex tool [12, 13] except that it is intended for the Scheme programming language [9]. In this paper, we will often refer to the article by Dubé and Feeley and the technique it describes as the "original paper" and the "original technique", respectively.

Sections 2 and 3 presents summaries of the original technique and SILex, respectively. Section 4 continues with a few definitions. Section 5 presents how we adapted the original technique so that it could fit into SILex. This section is the core of the paper. Section 6 quickly describes the changes that we had to make to SILex to add the new facility. Section 7 gives a few concrete examples of interaction with the new implementation. The speed of parse tree construction is evaluated in Section 8. Section 9 is a brief discussion about related work. Section 10 mentions future work.

2. Summary of the construction of parse tree for lexemes

Let us come back to the original technique. We just present a summary here since all relevant (and adapted) material is presented in details in the following sections.

The original technique aims at making lexical analyzers able to build parse trees for the lexemes that they match. More precisely, the goal is to make the *automatically generated* lexical analyzers able to do so. There is not much point in using the technique on analyzers that are written by hand. Note that the parse trees are those for the lexemes, not those for the regular expressions that match the latter. (Parse trees for the regular expressions themselves can be obtained using conventional syntactic analysis [1].) Such a parse tree is a demonstration of how a regular expression generates a word, much in the same way as a (conventional) parse tree demonstrates how a context-free grammar generates a word. Figure 1 illustrates what the parse trees are for a word `aab` that is generated by both a

* This work has been funded by the National Sciences and Engineering Research Council of Canada.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

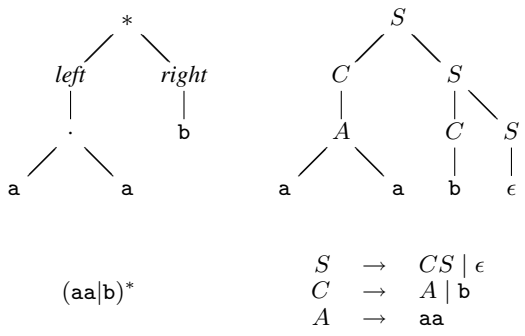


Figure 1. Parse trees for a word aab that is generated by a regular expression and a context-free grammar.

regular expression and an equivalent context-free grammar. While the parse tree on the right-hand side needs no explanation, the other one may seem unusual. It indicates the following: the Kleene star has been used for 2 iterations; in the first iteration, the *left*-hand side alternative has been selected and, in the second one, it was the *right*-hand side alternative; the sub-tree for aa is a concatenation (depicted by the implicit · operator). Note that the *left* and *right* labels are used for illustration purposes only. In general, any number of alternatives is allowed and the labels are numbered.

One might wonder why parse trees should be built for lexemes. Typically, compiler front-end implementors tend to restrict the lexical elements to relatively simple ones (e.g. identifiers, literal character constants, etc.). Even when more "complex" elements such as string constants are analyzed, it is relatively easy to write a decoding function that extracts the desired information from the lexemes. When some elements are genuinely more complex, their treatment is often deferred to the syntactic analysis. However, there are cases where the nature of the elements is truly lexical and where these are definitely *not* simple. In the introduction, we mentioned the numerical constants in Scheme. These are definitely lexical elements (no white space nor comments are allowed in the middle of a constant), yet their lexical structure is quite complex. In Section 7, we illustrate how one can benefit from obtaining parse trees for Scheme numerical constants. Moreover, it is a "chicken-and-egg" kind of issue since, by having more powerful tools to manipulate complex lexical elements, implementors may choose to include a wider variety of tasks as part of the lexical analysis phase.

The idea behind the technique described in the original paper is pretty simple. Because the automatically generated lexical analyzers are usually based on finite-state automata, the technique is based on automata too, but with a simple extension. The augmented automata are built using straightforward structural induction on the regular expressions to which they correspond. The addition to the automata consists only in putting *construction commands* on some arcs of the automata. The purpose of the construction commands is simple: let r be a regular expression, $A(r)$, the corresponding automaton, and w, a word; if a path P traverses $A(r)$ and causes w to be consumed, then the sequence of construction commands found along P forms a "recipe" that dictates how to build a parse tree t which is a demonstration that r generates w.

The automata that are built using the original technique are non-deterministic. It is well-known that performing lexical analysis using non-deterministic finite-state automata (NFA) is generally slower than using deterministic finite-state automata (DFA). Consequently, conversion of the NFA into DFA is desirable.

The augmented NFA can indeed be converted into DFA. However, note that a path taken through a DFA while consuming some word has little to do with the corresponding path(s) in the NFA, because of the presence of ϵ-transitions and arbitrary choices featured by the latter. Since one needs the construction commands of the NFA to build a parse tree, there must exist a mechanism that allows one to recover a path through the NFA from a path through the DFA. The technique proposes a mechanism that is implemented using three tables that preserve the connection between the DFA and the NFA. By making a series of queries to these tables, one is able to efficiently convert a path through the DFA into a corresponding path through the NFA. The path through the NFA trivially can be translated into a sequence on commands that explain how to build a parse tree. To summarize, the process of recognizing a lexeme and building a parse tree for it consists in identifying the lexeme using the DFA in the usual way while taking note of the path, recovering the path through the NFA, and then executing the sequence of commands.

The technique presented in the original paper deals only with the most basic regular operators: concatenation, union, and the Kleene star. Two distinct representations for the parse trees are introduced: the internal representation and the external one. The first one manipulates the trees as data structures. The second one manipulates them under their *printed* form, i.e. as words. Since the current paper is about lexical analyzers, we only consider the internal representation of parse trees. Finally, the original paper presents how one can obtain the complete set of parse trees for a word w that matches a regular expression r. Indeed, as shown below, the parse tree needs not be unique. In fact, there can be huge numbers (even an infinity) of parse trees, in some cases. Consequently, sets of parse trees are always represented under an implicit form only. We consider complete sets of parse trees to be mainly of theoretical interest and the current paper only considers the construction of a single parse tree for any lexeme.

3. SILex: a lexical analyzer generator for Scheme

SILex has originally been designed to be similar to the original Lex tool for the C language. In particular, the syntax of the regular expressions and the set of operators are the same. However, the actions that specify how to react to the recognition of a lexeme must be written in Scheme as expressions. In SILex, the actions *return* tokens while, in Lex, the actions produce tokens using a mixture of a returned value and side-effects. The third part of the specification files for Lex, which contains regular C code, does not have a counterpart in SILex. Consequently, the specification files for SILex include the part for the definition of macros (shorthands for regular expressions) and the part for the rules. SILex offers various services: many lexical analyzers may be used to analyze the same input; counters are automatically updated to indicate the current position inside of the input; the DFA can be represented using ordinary (compact) or portable tables, or can be directly implemented as Scheme code.

3.1 Lexemes

We describe the set $\mathcal{R}$ of regular expressions supported by SILex. All regular expressions in $\mathcal{R}$ are presented in Figure 2. Each kind of regular expression is accompanied by a short description and its language. We use Σ to denote the set of characters of the Scheme implementation at hand (e.g. the ASCII character set). The language of a regular expression r is denoted by $L(r)$. In the figure, c ranges over characters ($c \in \Sigma$), i and j ranges over integers ($i, j \in \mathbb{N}$), *spec* denotes the specification of the contents of a character class, C ranges over character classes ($C \subseteq \Sigma$), and v ranges over strings ($v \in \Sigma^*$). All variables r and r_i are assumed to be in $\mathcal{R}$. Finally, $\rho_L : \mathcal{R} \times \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \to 2^{\Sigma^*}$ is a repetition function defined

DESCRIPTION	REGULAR EXPRESSION	LANGUAGE
Ordinary character	c	$\{c\}$
Any character	.	$\Sigma - \{\textit{newline character}\}$
Newline character	\n	$\{\textit{newline character}\}$
Character by code	$\backslash i$	$\{\textit{character of code } i\}$
Quoted character	$\backslash c$	$\{c\}$
Character class	$[spec]$	$C \subseteq \Sigma$
Literal string	"v"	$\{v\}$
Parenthesized expression	(r)	$L(r)$
Kleene closure	r^*	$\rho_L(r, 0, \infty)$
Positive closure	r^+	$\rho_L(r, 1, \infty)$
Optional expression	$r^?$	$\rho_L(r, 0, 1)$
Fixed repetition	$r\{i\}$	$\rho_L(r, i, i)$
At-least repetition	$r\{i,\}$	$\rho_L(r, i, \infty)$
Between repetition	$r\{i, j\}$	$\rho_L(r, i, j)$
Concatenation	$r_0 \dots r_{n-1}$	$L(r_0) \dots L(r_{n-1})$
Union	$r_0 \mid \dots \mid r_{n-1}$	$L(r_0) \cup \dots \cup L(r_{n-1})$

Figure 2. Regular expressions supported by SILex and the corresponding languages.

as:

$$\rho_L(r, b, B) = \bigcup_{\substack{i \in \mathbb{N} \\ b \leq i \leq B}} (L(r))^i$$

Many details are omitted in the presentation of $\mathcal{R}$ by lack of relevance for this paper. For instance, the exact set of ordinary characters and the syntax of the character classes are not really interesting here. For complete information about the syntax, we refer the reader to the documentation of SILex [3]. The important thing to know about character classes is that an expression $[spec]$ matches nothing else than a single character and it does match a character c if $c \in C$ where C is the set of characters denoted by $spec$.

Operators used to build up regular expressions have different priority. We assume that the repetition operators (*, $^?$, $\{i,\}$, ...) have higher priority than the (implicit) concatenation operator and that the latter has higher priority than the union operator. Moreover, we expect unions (and concatenations) to account for all subexpressions that are united (or concatenated, respectively). In other words, when we write a union $r_0 \cup \dots \cup r_{n-1}$, none of the r_i should be a union. Likewise, when we write a concatenation $r_0 \dots r_{n-1}$, none of the r_i should be a concatenation (nor a union, naturally). Repetition operators, though, can be piled up (e.g. as in expression $\mathtt{d}^?\{2,4\}^+$).

From now on, we forget about the first 5 kinds of regular expressions. These can all be represented by totally equivalent character classes (equivalent according to their language *and* according to their associated parse trees, too). For instance, expressions `f` and . can be replaced by [f] and [^\n], respectively. As for the literal strings, we choose *not* to forget about them. Although it could be tempting to replace them by concatenation of characters, which would denote the same language, we refrain to do so because, as we see later, it would change the associated parse trees. For efficiency reasons, the parse trees for literal strings are different from those for concatenations. The former are cheaper to generate than the latter.

3.2 Incompatibilities with the original technique

The original technique for the construction of parse trees for lexemes cannot be integrated directly into SILex for two reasons. First, SILex provides a larger set of operators in regular expressions than the one presented in the original paper. Second, the original technique builds lists by adding elements *to the right*. This does not correspond to the efficient and purely functional way of building lists in Scheme. Consequently, the rules for the construction of the NFA with commands have to be adapted to the larger set of operators and to the direction in which Scheme lists are built.

4. Definitions

There are some terms specific to the domain of lexical analysis that need to be defined. At this point, we have already defined *regular expressions* along with their language. In the context of compiler technology, unlike in language theory, we are not only interested in checking if a word w matches a regular expression r (i.e. whether $w \in L(r)$), but also in the decomposition of the input u ($\in \Sigma^*$) into a stream of lexemes that leads to a stream of tokens. A *lexeme* is a prefix w of the input u ($u = wu'$) that matches some regular expression r. Based on the matching regular expression r and the matched lexeme w, a *token* is produced. Examples of tokens include: the identifier named `trace`, the reserved keyword `begin`, the operator `+`, etc. Typically, the stream of tokens that is produced by lexical analysis constitutes the input to the syntactic analyzer. While the concept of token is variable and depends on the application, the concept of lexeme is standard and can be defined in terms of language theory. Usually, when a lexeme w has been identified, i.e. when $u = wu'$, and that the corresponding token has been produced, w is considered to have been consumed and the remaining input is u'.

In the context of automatic generation of lexical analyzers, there is typically more than one regular expression, r_i, that may match lexemes. Lexical analyzers are usually specified using a list of *rules*, each rule being an association between a regular expression r_i and an *action* α_i. An action α_i is some statement or expression in the target programming language that indicates how to produce tokens when lexemes are found to match r_i. The action normally has access to the matching lexeme and also has the opportunity to create some side effects such as: updating the table of symbols, increasing counters, etc. During lexical analysis, the analyzer may match a prefix of the input with the regular expression r_i of any (active) rule.

Lexical analyzers produced by SILex, like many other lexical analyzers, obey some principles when trying to find and select matches. SILex follows the *maximal-munch* (aka, longest-match) *tokenization* principle. It means that when there is a match between prefix w_1 and regular expression r_i that compete with another match between prefix w_2 and expression r_j, such that $|w_1| > |w_2|$,

$$
\begin{aligned}
T([spec], w) &= \begin{cases} \{w\}, & \text{if } w \in L([spec]) \\ \emptyset, & \text{otherwise} \end{cases} \\
T(\text{"}v\text{"}, w) &= \begin{cases} \{v\}, & \text{if } w = v \\ \emptyset, & \text{otherwise} \end{cases} \\
T((r), w) &= T(r, w) \\
T(r^*, w) &= \rho_T(r, w, 0, \infty) \\
T(r^+, w) &= \rho_T(r, w, 1, \infty) \\
T(r^?, w) &= \rho_T(r, w, 0, 1) \\
T(r\{i\}, w) &= \rho_T(r, w, i, i) \\
T(r\{i,\}, w) &= \rho_T(r, w, i, \infty) \\
T(r\{i, j\}, w) &= \rho_T(r, w, i, j) \\
T(r_0 \dots r_{n-1}, w) &= \left\{ [t_0, \dots, t_{n-1}] \;\middle|\; \begin{array}{l} \exists w_0 \in \Sigma^*. \ \dots \ \exists w_{n-1} \in \Sigma^*. \\ w = w_0 \dots w_{n-1} \wedge \\ \forall 0 \le i < n.\ t_i \in T(r_i, w_i) \end{array} \right\} \\
T(r_0 \mid \dots \mid r_{n-1}, w) &= \{\#i : t \mid 0 \le i < n \ \wedge \ t \in T(r_i, w)\}
\end{aligned}
$$

where:

$$
\rho_T(r, w, b, B) = \left\{ [t_0, \dots, t_{n-1}] \;\middle|\; \begin{array}{l} \exists n \in \mathbb{N}.\ b \le n \le B \wedge \\ \exists w_0 \in \Sigma^*. \ \dots \ \exists w_{n-1} \in \Sigma^*. \\ w = w_0 \dots w_{n-1} \wedge \\ \forall 0 \le i < n.\ t_i \in T(r, w_i) \end{array} \right\}
$$

Figure 3. Parse trees for a word that matches a regular expression.

then the former match is preferred. SILex also gives priority to first rules. It means that when there is a match between prefix w and expression r_i that compete with another match between w and r_j, such that $i < j$, then the former match is preferred. Note that, although these two principles uniquely determine, for each match, the length of the lexeme and the rule that matches, they say nothing about the parse tree that one obtains for the lexeme. As we see below, a single pair of a regular expression and a word may lead to more than one parse tree. In such a case, the lexical analyzer is free to return any of these.

5. Adapting the construction of parse trees

Before the adapted technique is presented, the notation for the parse trees is introduced and the parse trees for a word according to a regular expression. The following two subsections present the finite-state automata that are at the basis of the construction of parse trees. Finally, we consider the issue of converting the NFA into DFA.

5.1 Syntax of the parse trees

Let us present the syntax of the parse trees. Let $\mathcal{T}$ be the set of all possible parse trees. $\mathcal{T}$ contains basic trees, which are words, and composite trees, which are *selectors* and lists. $\mathcal{T}$ is the smallest set with the following properties.

$$
\begin{array}{lll}
\forall w \in \Sigma^*. & & w \in \mathcal{T} \\
\forall i \in \mathbb{N}. \quad \forall t \in \mathcal{T}. & & \#i : t \in \mathcal{T} \\
\forall n \ge 0. \quad \forall i \in \mathbb{N} \text{ s.t. } 0 \le i < n. & \forall t_i \in \mathcal{T}. & \\
& [t_0, \dots, t_{n-1}] \in \mathcal{T} &
\end{array}
$$

Note that we do not represent parse trees graphically as is customary in presentation of parsing technology. Instead, we use a notation similar to a data structure (to an algebraic data type, to be more specific) to represent them. However, the essence of both representations is the same as the purpose of a parse tree is to serve as an explicit demonstration that a particular word can effectively be generated by a regular expression (or, usually, by a context-free grammar).

In particular, let us recall that if we have a parse tree t for a word w according to a context-free grammar, then we can find all the characters of w, in order, at the leaves of t. We can do the same with our parse trees associated to regular expressions. Let us define an extraction function $X : \mathcal{T} \to \Sigma^*$ that allows us to do so.

$$
\begin{aligned}
X(w) &= w \\
X(\#i : t) &= X(t) \\
X([t_0, \dots, t_{n-1}]) &= X(t_0) \dots X(t_{n-1})
\end{aligned}
$$

5.2 Parse trees for lexemes

We can now describe the parse trees for a word that matches a regular expression. Figure 3 presents the T function. $T(r, w)$ is the set of parse trees that show how w is generated by r. We use the plural form "parse trees" as there may be more than one parse tree for a single expression/word pair. Borrowing from the context-free grammar terminology, we could say that a regular expression may be *ambiguous*.

Note that, once again, we need a repetition function $\rho_T : \mathcal{R} \times \Sigma^* \times \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \to 2^{\mathcal{T}}$ to help shorten the definitions for the numerous repetition operators. The definition of the repetition function can be found at the bottom of Figure 3.

The meaning of $T(r', w)$, for each form of r', is explained in the following. Some examples are given. Note that, for the sake of brevity, we may use single-character regular expressions such as `a` instead of the equivalent class variants such as `[a]`.

- Case $r' = [spec]$. The only valid parse tree, if it exists, is a single character c. c has to be a member of the character class specification and has to be equal to the single character in w. Examples: $T([\texttt{ab}], \texttt{a}) = \{\texttt{a}\}$; $T([\texttt{ab}], \texttt{c}) = \emptyset = T([\texttt{ab}], \texttt{baa})$.

- Case $r' = "v"$. The only valid parse tree is v and it exists if $w = v$. Note that, from the point of view of the parse tree data type, parse tree v is considered to be atomic (or basic), even though, from the point of view of language theory, $v \in \Sigma^*$ may be a composite object. Example: $T(\texttt{"abc"}, \texttt{abc}) = \{\texttt{abc}\}$.
- Case $r' = (r)$. Parentheses are there just to allow the user to override the priority of the operators. They do not have any effect on the parse trees the are generated.
- Cases $r' = r^*$, $r' = r^+$, $r' = r^?$, $r' = r\{i\}$, $r' = r\{i,\}$, and $r' = r\{i,j\}$. The parse trees for w demonstrate how w can be partitioned into n substrings w_0, ..., w_{n-1}, where n is legal for the particular repetition operator at hand, and how each w_i can be parsed using r to form a child parse tree t_i, with the set of all the t_i collected into a list. The lists may have varying lengths but the child parse trees they contain are all structured according to the single regular expression r. Example: $T(\texttt{a}\{2,3\}^*, \texttt{aaaaaa}) = \{[[\texttt{a}, \texttt{a}], [\texttt{a}, \texttt{a}], [\texttt{a}, \texttt{a}]], [[\texttt{a}, \texttt{a}, \texttt{a}], [\texttt{a}, \texttt{a}, \texttt{a}]]\}$.
- Case $r' = r_0 \ldots r_{n-1}$. The parse trees for w demonstrate how w can be partitioned into exactly n substrings w_0, ..., w_{n-1}, such that each w_i is parsed according to its corresponding child regular expression r_i. In this case, the lists have constant length but the child parse trees are structured according to various regular expressions. Examples: $T(\texttt{abc}, \texttt{abc}) = \{[\texttt{a}, \texttt{b}, \texttt{c}]\}$; $T(\texttt{a}^*\texttt{ab}, \texttt{aaab}) = \{[[\texttt{a}, \texttt{a}], \texttt{a}, \texttt{b}]\}$.
- Case $r' = r_0 \mid \ldots \mid r_{n-1}$. A parse tree for w demonstrates how w can be parsed according to one of the child regular expressions. It indicates which of the child expressions (say r_i) matched w and it contains an appropriate child parse tree (for w according to r_i). Example: $T(\texttt{a}^*|(\texttt{aa})^+|\texttt{a}^?\texttt{a}^?, \texttt{a}) = \{\#0 : [\texttt{a}], \#2 : [[\texttt{a}], [\,]], \#2 : [[\,], [\texttt{a}]]\}$.

Function T has some interesting properties. The first one is that parse trees exist only for words that match a regular expression; formally, $T(r, w) \neq \emptyset$ if and only if $w \in L(r)$. The second one is that, from any parse tree for a word according to a regular expression, we can extract the word back; formally, if $t \in T(r, w)$, then $X(t) = w$.

Depending on the regular expression, the "amount" of ambiguity varies. The union operator tends to additively increase the number of different parse trees produced by the child expressions. On the other hand, the concatenation operator tends to polynomially increase the number of different parse trees. Even more extreme, some of the repetition operators tend to increase the number exponentially and even infinitely. Let us give instances of such increases. Let the r_i's be expressions that lead to one or two parse trees for any non-empty word and none for ϵ. Then $r_0 \mid \ldots \mid r_{n-1}$, $r_0 \ldots r_{n-1}$, $((r_0)^+)^*$, and $((r_0)^*)^*$ produce additive, polynomial, exponential, and infinite increases, respectively.

5.3 Strategy for the construction of parse trees

In the original paper, it is shown how the construction of parse trees for lexemes can be automated. The technique is an extension of Thompson's technique to construct finite-state automata [14]. The extension consists in adding *construction commands* on some of the edges of the automata. Essentially, each time a path through an automaton causes some word to be consumed, then the sequence of commands found along that path forms a "recipe" for the construction of a parse tree for the word.

In general, a parse tree may be an assemblage of many subtrees. These sub-trees cannot all be built at once. They are created one after the other. Consequently, the sub-trees that are already built have to be kept somewhere until they are joined with the other subtrees. It was shown that a data structure as simple as a stack was providing the appropriate facilities to remember and give back parts of a parse tree under construction. All the parse tree construction commands are meant to operate on a stack.

The commands used by the original technique are: "push constant", "wrap in selector", and "extend list". The "push constant" command has a constant tree t as operand and performs the following operation: it modifies the stack it is given by pushing t. The "wrap in selector" command has a number i as operand and performs the following operation: it modifies the stack by first popping a tree t, by building the selector $\#i : t$, and then by pushing $\#i : t$ back. Finally, the "extend list" command has no operand and performs the following operation: it modifies the stack by first popping a tree t and then a list l, by adding t *at the end of* l to form l', and then by pushing l' back.

As explained above, the Scheme language does feature lists but these lists are normally (efficiently) accessed by the front and not by the end. Strictly speaking, Scheme lists *can* be extended efficiently by the end but only in a destructive manner. We prefer to avoid going against the usual programming style used in functional languages and choose to adapt the original technique to make it compatible with the natural right to left construction of lists in Scheme.

This choice to adapt the original technique to build lists from right to left has an effect on the way automata with commands are traversed. In the adapted technique, we have the property that, if a path traverses an automaton forwards and consumes some word, then the sequence of commands found *on the reversed path* forms a recipe to build a parse tree for the word. Thus, the next section presents a technique to build finite-state automata with commands similar to that of the original paper except for the facts that we have a larger set of regular expression operators and that the commands are placed differently in the automata.

5.4 Automata with construction commands

We present the construction rules for the finite-state automata with commands. The construction rules take the form of a procedure A that takes a regular expression r and builds the automaton $A(r)$. A is defined by structural induction on regular expressions. The construction rules are similar to those prescribed by Thompson [14] but with commands added on the edges. The rules are presented in Figures 4 and 5.

Each construction rule produces an automaton with distinguished entry and exit states named p and q, respectively. When an automaton $A(r)$ embeds another one $A(r')$, we depict $A(r')$ as a rectangle with two states which are the entry and exit states of $A(r')$. In each automaton $A(r)$, there is no path going from q to p using edges of $A(r)$ only. In other words, any path from q to p, if it exists, has to go through at least one edge added by a surrounding automaton. The parse tree construction commands are shown using a compact notation. A "push constant" command with operand t is denoted by `push` t. A "wrap in selector" command with operand i is denoted by `sel` i. An "extend list" command is (of course) denoted by `cons`.

We mention, without proof, the few key properties of the automata. Let r be a regular expression and P be a path that traverses $A(r)$ from entry to exit. First, the sequence of commands that are met by following P *backwards* causes exactly one parse tree to be pushed. More precisely, if we take a stack σ and apply on it all the commands that we meet by following P *backwards*, then the net effect of these commands is to push exactly one parse tree t on σ. Second, the automata are correct in the sense that if the word that is consumed along P is w, then $t \in T(r, w)$. Third, the automata are exhaustive with respect to T in the sense that, for any $r \in \mathcal{R}$, $w \in L(r)$, $t \in T(r, w)$, and stack σ, then there exists a path P that traverses $A(r)$, that consumes w, and whose reversed sequence of

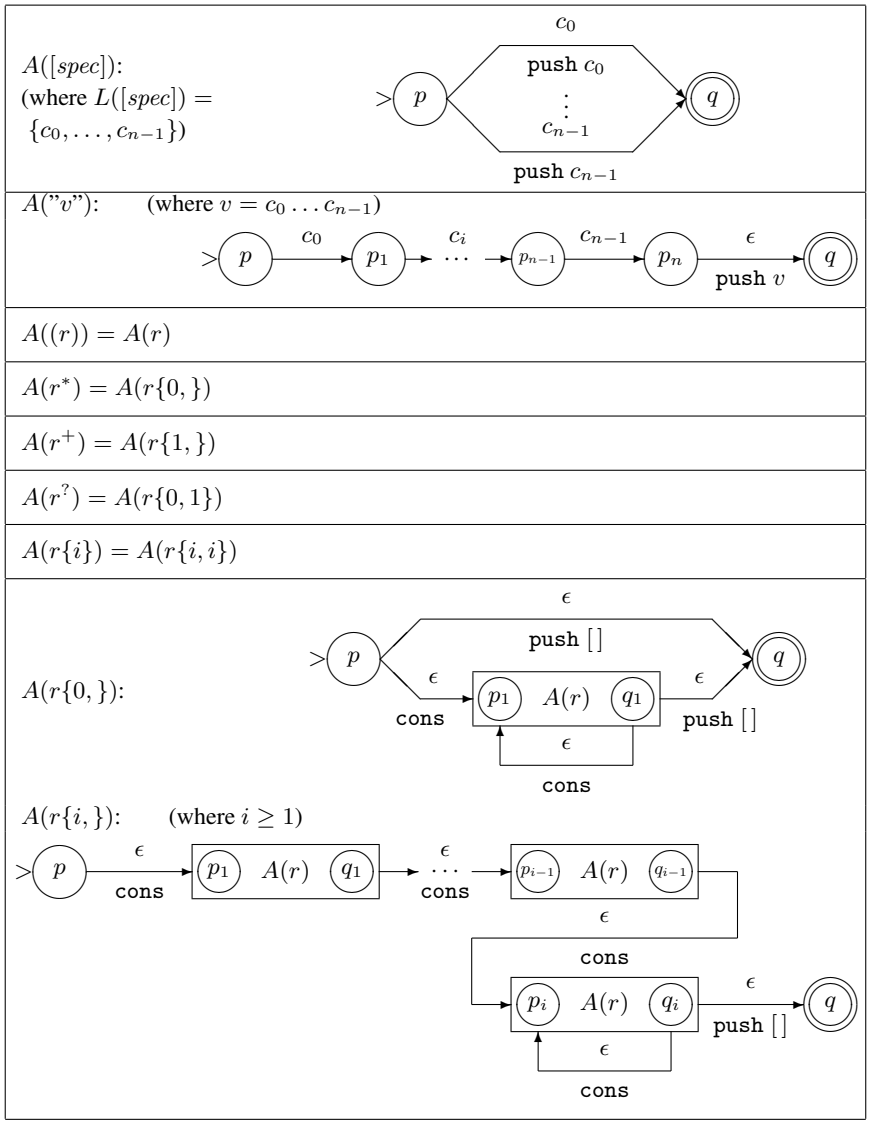


Figure 4. Construction rules for the automata with commands (Part I).

commands causes t to be pushed on σ. These properties can be proved straightforwardly by structural induction on $\mathcal{R}$.

5.5 Using deterministic automata

For efficiency reasons, it is preferable to use a DFA instead of a NFA. As explained above, the NFA obtained using function A may be converted into a DFA to allow fast recognition of the lexemes but three tables have to be built in order to be able to translate paths through the DFA back into paths through the original NFA.

We assume the conversion of the NFA into a DFA to be a straightforward one. We adopt the point of view that deterministic states are sets of non-deterministic states. Then, our assumption says that the deterministic state that is reached after consuming some word w is *exactly* the set of non-deterministic states that can be reached by consuming w.[1]

We may now introduce the three tables Acc, f, and g. Table g indicates how to reach a state q from the non-deterministic start state using only ϵ-transitions. It is defined only for the non-deterministic states that are contained in the deterministic start state. Table f indicates how to reach a non-deterministic state q from some state in a deterministic state s using a path that consumes a single character c. It is usually not defined everywhere. Table Acc indicates, for a deterministic state s, which non-deterministic state in s accepts on behalf of the same rule as s. It is defined only for accepting deterministic states.

Let us have a word $w = c_0 \dots c_{n-1}$ that is accepted by the DFA and let $P_D = s_0 \dots s_n$ be the path that is taken when w is consumed. Each s_i is a deterministic state, s_0 is the start state, and s_n is an accepting state. Note that an accepting state does not simply accept, but it accepts on behalf of a certain rule. In fact, an accepting deterministic state may contain more than one accepting

[1] Note that this assumption precludes full minimization of the DFA. SILex currently does not try to minimize the DFA it builds. The assumption is sufficiently strong to ensure that paths through the NFA can be recovered but it may happen to be unnecessarily strong. More investigation should be made to find a sufficient and necessary condition on the conversion.

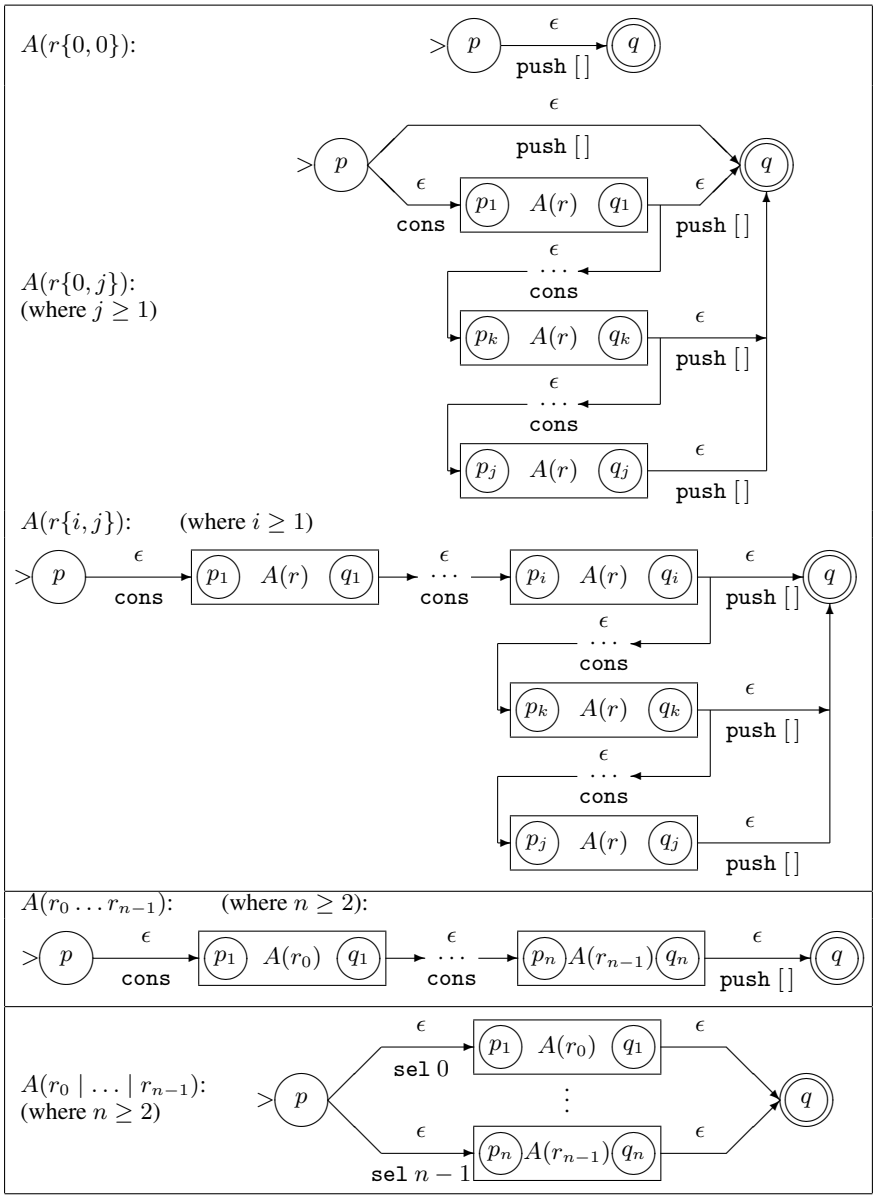


Figure 5. Construction rules for the automata with commands (Part II).

non-deterministic states, each on behalf of its corresponding rule. In such a case, the deterministic state accepts on behalf of the rule that has highest priority. The non-deterministic path P_N that corresponds to P_D is recovered backwards portion by portion. The idea consists in determining non-deterministic states $\{q_i\}_{0 \leq i \leq n}$ and portions of path $\{P_i\}_{0 \leq i \leq n}$ such that: each q_i is in s_i; each P_i starts at q_{i-1}, ends at q_i, and consumes c_{i-1}, except for P_0, which starts at the non-deterministic start state, ends at q_0, and consumes ϵ; q_n is a state that accepts on behalf of the same rule as s_n.

The recovery is initialized by determining q_n directly from s_n using the query $Acc(s_n)$. Next, the main part of the recovery consists in an iteration, with i going from n down to 1. At step i, given q_i, one can determine portion of path P_i and intermediate non-deterministic state q_{i-1}. P_i is obtained from the query $f(s_{i-1}, c_{i-1}, q_i)$. By doing so, q_{i-1} is also obtained as it is the source state of P_i. As the final part of the recovery, P_0 is obtained using the query $g(q_0)$. Then path P_N is simply the linkage of all the portions together; i.e. $P_N = P_0 \cdot \ldots \cdot P_n$.

Note that the preceding explanation contains some minor inaccuracies. First, tables f and g do not exactly contain portions of path but *reversed* ones. Indeed, recall that the NFA presented in this paper are such that commands must be executed in the order in which they are met when following paths backwards. Second, there is no need to recover path P_N (or its reverse) explicitly. It is sufficient to keep references to the portions that form P_N and to later execute the commands by following the portions one after the other. Better yet, one may eagerly execute the commands contained in each portion as the latter gets determined. This way, it is unnecessary to remember P_N nor its portions. Only the current state of the construction stack needs to be preserved. Last, one may observe that the sole purpose of the portions of path stored in tables f and g is to be followed in order to recover the parse tree

construction commands. It is possible to skip the step of converting a portion of path into a sequence of commands by directly storing sequences of commands in f and g. It not only saves time by avoiding the conversion but also because sequences of commands can be no longer than the paths from which they are extracted since at most one command gets attached to each arc. One must be careful in the case of table f because a mere sequence of commands would not indicate which non-deterministic state is the origin of the portion of path. Consequently, the latter also has to be returned by f. To recapitulate: a query $g(q)$ provides the sequence of commands that would be met by following some ϵ-consuming path from the non-deterministic start state to q backwards; a query $f(s, c, q)$ provides a pair of a non-deterministic state $q' \in s$ and the sequence of commands that would be met by following some c-consuming path from q' to q backwards.

Remember that some regular expressions are ambiguous. Let r be an ambiguous expression and w a word that has at least two parse trees. We said that, in the context of automatically generated lexical analyzers, it is sufficient to build only one parse tree for w. In other words, from the path P_D that traverses the DFA, it is sufficient to recover only one (P_N) of the corresponding paths that traverse the NFA. Indeed, by the use of fixed tables f, g, and Acc, the recovery of P_N from P_D and w is deterministic. Essentially, the choices among all possible paths are indirectly made when unique values are placed into tables entries that could have received any of numerous valid values. Nevertheless, even if in practice, any single lexical analyzer produces parse trees in a deterministic manner, it remains more convenient to specify the parse tree construction as a non-deterministic process.

6. Modifications to SILex

The addition of parse trees to SILex has little impact on the way SILex is used. The only visible modification is the presence of an additional variable in the scope of the actions. The name of this variable is `yyast`, for Abstract Syntax Tree.[2] An action may refer to this variable as any other variable provided by SILex, such as `yytext`, which contains the lexeme that has just been matched, `yyline`, which contains the current line number, etc.

While the observable behavior of SILex has not changed much, there are many changes that have been made to the implementation of SILex. The most important changes were made in the generation-time modules. First, the original version of SILex used to convert many regular expression operators into simpler forms in order to handle as few native operators as possible. It was doing so during syntactic analysis of the regular expressions. For example, SILex eliminated some forms by converting strings like "v" into concatenations, by breaking complex repetition operators into a combination of simpler ones and concatenations, and by splitting large concatenations and unions into binary ones. While such conversions do not change the language generated by the expressions, they *do* change the set of valid parse trees for most or all words. The new version has to represent most syntactic forms as they appear in the specification files. Still, there are now new opportunities to translate simple forms, such as r^*, r^+, $r^?$, and $r\{i\}$, into the more general forms $r\{b, B\}$, which have to be supported anyway.

Second, the construction rules for the automata have been changed to correspond to the new list of syntactic forms and to conform to the specifications of Figures 4 and 5. Of course, the representation of the arcs (in the NFA) had to be extended so that commands could be attached.

[2] Actually, we consider the name `yyast` to be rather inappropriate as the parse trees that the new variable contains are indeed *concrete* syntax trees. Still, since the version of SILex that we are working on uses that name, we prefer to stick to the current conventions.

Third, a phase which used to clean up the NFA between the elimination of the ϵ-transitions and the conversion of the NFA into a DFA has been eliminated. It eliminated useless states and renumbered the remaining states. The modification of the numbers interfered with the construction of the three new tables and the quick and dirty solution has been to completely omit the phase. The generated analyzers would benefit from the re-introduction of the clean-up phase and, in order to do so, some adaptation should be made to the currently abandoned phase or to the implementation of the table construction.

Fourth, we added the implementation of the construction and the printing of the three tables. The construction of the tables mainly consists in extracting reachability information from the graph of the NFA.

The next modifications were made to the analysis-time module. Fifth, the lexical analyzers had to be equipped with instrumentation to record the paths that are followed in the DFA. Also, requests for the construction of parse trees when appropriate have been added.

Sixth, we included the functions that build parse trees when they are given a path through the DFA, the recognized lexeme, and the three tables.

Up to this point, the modifications aimed only at providing the parse tree facility when the tables of the DFA are represented using the ordinary format. So, at last, we modified both the generation-time and the analysis-time modules so that parse trees could also be built when the DFA is represented using portable tables or Scheme code. In the case of the portable tables, it required only the creation of simple conversion functions to print a portable version of tables f and g at generation time and to translate the portable tables back into the ordinary format at analysis time. In the case of the DFA as Scheme code, the modifications are more complex as extra code must be emitted that takes care of the recording of the path through the DFA and the requests for the construction of parse trees. Note that the functions that perform the very construction of the parse trees are the same no matter which format for the tables of the DFA is used. It means that the construction of parse trees is an interpretative process (based on queries to the three tables), even when the DFA is implemented efficiently as code.

Note that, although SILex gives the impression that parse trees are always available to actions, SILex is lazy with their construction. It builds them only for the actions that *seem* to access the variable `yyast`. The path followed into the DFA is always recorded, however. Still, SILex's laziness substantially reduces the extra cost caused by the addition of the parse trees as most of it comes from the construction of trees, not the recording of paths.

The current state of the prototype is the following. The integration is complete enough to work but the code needs a serious clean-up. The three additional tables for DFA to NFA correspondence are much too large. The implementation of the mechanisms for path recording and parse tree construction is not really optimized for speed.

7. Examples of parse tree construction for lexemes

We present a few concrete examples of the use of parse tree construction using SILex. We first start by describing the Scheme representation of the parse trees.

7.1 Representation of parse trees in Scheme

The representation of trees in $\mathcal{T}$ in Scheme is direct. A list tree $[t_0, \ldots, t_{n-1}]$ becomes a Scheme list `(`S_0 ... S_{n-1}`)` where each S_i is the Scheme representation of t_i. Next, a selector $\#i : t$ also becomes a Scheme list `(`i S`)` where i remains the same and S corresponds to t. Finally, a word w may take two forms in Scheme.

If w is a parse tree that originates from a string regular expression "w", then it becomes a Scheme string "w", otherwise w is necessarily one-character long and it becomes a Scheme character #\w.

7.2 Simple examples

Let us consider the following short SILex specification file:

```
%%
a{2,4}  (list 'rule1 yyast)
a{0,3}  (list 'rule2 yyast)
```

where only some sequences of `a` are deemed to be legal tokens and where the actions simply return tagged lists containing the parse trees that are produced. If we generate a lexical analyzer from this specification file and ask it to analyze the input `aaaaa`, then it will produce the following two results before returning the end-of-file token:

```
(rule1 (#\a #\a #\a #\a))
(rule2 (#\a))
```

Both parse trees indicate that the matched lexemes were made of repetitions of the character `a`, which is consistent with the shape of the regular expressions. Note how the first token had to be as long as possible, following the maximal-munch tokenization principle.

Now, let us consider a more complex example. The following specification file allows the analyzer-to-be to recognize Scheme strings:

```
%%
\"([^"\\]|"\\""|"\\\\")*\"  yyast
```

One must not forget about the necessary quoting of special characters " and \. If we feed the analyzer generated from this specification with the following Scheme string:

```
"Quote \" and \\!"
```

then the analyzer returns a parse tree that denotes a sequence of three sub-trees, where the middle one is a sequence of 14 sub-sub-trees, where each is a selector among the three basic string elements:

```
(#\"
 ((0 #\Q) (0 #\u) (0 #\o) (0 #\t) (0 #\e)
  (0 #\space) (1 "\\\"") (0 #\space)
  (0 #\a) (0 #\n) (0 #\d) (0 #\space)
  (2 "\\\\") (0 #\!))
 #\")
```

These two examples may not be that convincing when it comes to justifying the implementation of automatic construction of parse trees for lexemes. However, the one below deals with a regular expression that is way more complex.

7.3 Lexical analysis of Scheme numbers

Scheme provides a particularly rich variety of numbers: from integers to complex numbers. It also provides a "syntax" for the external representation of all these kinds of numbers. An implementor has much work to do in order to handle all the kinds of numbers. In particular, when it comes to *reading* them. There are so many cases that reading them in an *ad hoc* way tends to be error-prone.

Even when one automates part of the process by using an automatically generated lexical analyzer to scan Scheme numbers, only half of the problem is solved. Indeed, merely knowing that a lexeme is the external representation of a Scheme number does not provide any easy way to recover the internal representation from the lexeme. That is, it is not easy unless the lexical analyzer is able to provide a parse tree for the lexeme. In Figure 6, we present a relatively complete specification for the Scheme numbers. Note that we restrict ourselves to numbers in base 10 only and that we do not handle the unspecified digits denoted by #. The specification file is mostly made of macros and there is a single rule which takes the parse tree for the number and passes it to a helper function.

The helper function is very simple as it only has to traverse the tree and *rebuild* the number. This reconstruction is made easy by the fact that the hierarchical structure of the lexeme according to the regular expression is clearly exposed and that any "choice" between various possibilities is indicated by the tree. Figure 7 presents the implementation of our helper function, which is less than one hundred lines of very systematic code. The reader needs not necessarily study it closely—the font is admittedly pretty small—as the main point here is to show the size and the shape of the code. If we were to complete our implementation to make it able to handle the full syntax, it would be necessary to add many macros in the specification file but the helper function would not be affected much.

8. Experimental results

In order to evaluate the cost of the construction of parse trees, we ran a few experiments. The experiments consist in analyzing the equivalent of 50 000 copies of the following 10 numbers (as if it were a giant 500 000-line file).

```
32664
-32664
32664/63
+32664/63
-98327E862
+i
-453.3234e23+34.2323e1211i
+.32664i
-3266.4@63e-5
+32664/63@-7234.12312
```

We used three different lexical analyzers on the input. The first one is a lexical analyzer generated by the original version of SILex. The second one is generated by the new version of SILex and build a parse tree for each of the recognized lexemes. The third one is also generated using the new version of SILex but it does not ask for the construction of the parse trees (i.e. the action does not access `yyast`). This last analyzer is used to evaluate the cost of the instrumentation added to record the path through the DFA.

The lexical analyzers have been generated by (either version of) SILex to be as fast as possible; that is, their DFA is implemented as Scheme code and they maintain no counters to indicate the current position in the input. The lexical analyzers have been compiled using Gambit-C version 3.0 with most optimizations turned on. The resulting C files have been compiled using GCC version 3.3.5 with the '`-O3`' switch. The analyzers were executed on a 1400 MHz Intel Pentium 4 processor with 512 MBytes of memory.

The execution times for the three analyzers are 15.3 seconds, 39.8 seconds, and 20.2 seconds, respectively. Clearly, building the parse trees incurs a serious cost as the execution time almost triples. This is not that surprising given the complexity of building a parse tree compared to the simplicity of a mere recognition using a DFA. However, the third measurement indicates that the added instrumentation causes the operations of the DFA to take significantly longer. The increase is about by a third. While the increase is much less than in the case of parse tree construction, it is still less acceptable. Construction of parse trees can be seen as a sophisticated operation that is relatively rarely performed. One might accept more easily to pay for a service that he does use. However, the extra cost due to the instrumentation is a cost without direct benefit and that

```
; Regular expression for Scheme numbers
; (base 10 only, without '#' digits)

digit           [0-9]
digit10         {digit}
radix10         ""|#[dD]
exactness       ""|#[iI]|#[eE]
sign            ""|"+"|"-"
exponent_marker [eEsSfFdDlL]
suffix          ""|{exponent_marker}{sign}{digit10}+
prefix10        {radix10}{exactness}|{exactness}{radix10}
uinteger10      {digit10}+
decimal10       ({uinteger10}|"."{digit10}+|{digit10}+"."{digit10}*){suffix}
ureal10         {uinteger10}|{uinteger10}/{uinteger10}|{decimal10}
real10          {sign}{ureal10}
complex10       {real10}|{real10}@{real10}|{real10}?[-+]{ureal10}?[iI]
num10           {prefix10}{complex10}
number          {num10}

%%

{number}        (lex-number yyast)
```

Figure 6. SILex specification for the essentials of the lexical structure of Scheme numbers.

```
; Companion code for Scheme numbers

(define lex-number
  (lambda (t)
    (let* ((digit
            (lambda (t)
              (- (char->integer t) (char->integer #\0))))
           (digit10
            (lambda (t)
              (digit t)))
           (exactness
            (lambda (t)
              (case (car t)
                ((0)  (lambda (x) x))
                ((1)  (lambda (x) (* 1.0 x)))
                (else (lambda (x) (if (exact? x) x (inexact->exact x)))))))
           (sign
            (lambda (t)
              (if (= (car t) 2)
                  -1
                  1)))
           (digit10+
            (lambda (t)
              (let loop ((n 0) (t t))
                (if (null? t)
                    n
                    (loop (+ (* 10 n) (digit10 (car t))) (cdr t))))))
           (suffix
            (lambda (t)
              (if (= (car t) 0)
                  0
                  (let ((tt (cadr t)))
                    (* 1.0
                       (sign (list-ref tt 1))
                       (digit10+ (list-ref tt 2)))))))
           (prefix10
            (lambda (t)
              (exactness (list-ref (cadr t) (- 1 (car t))))))
           (uinteger10
            (lambda (t)
              (digit10+ t)))
           (decimal10
            (lambda (t)
              (let* ((e2 (suffix (list-ref t 1)))
                     (tt (list-ref t 0))
                     (ttt (cadr tt)))
                (case (car tt)
                  ((0)
                   (* (digit10+ ttt) (expt 10 e2)))
                  ((1)
                   (let ((tttt (list-ref ttt 1)))
                     (* (digit10+ tttt) (expt 10.0 (- e2 (length tttt))))))
                  (else
                   (let* ((tttt1 (list-ref ttt 0))
                          (tttt2 (list-ref ttt 2)))
                     (* (digit10+ (append tttt1 tttt2))
                        (expt 10.0 (- e2 (length tttt2))))))))))
           (ureal10
            (lambda (t)
              (let ((tt (cadr t)))
                (case (car t)
                  ((0)
                   (uinteger10 tt))
                  ((1)
                   (/ (uinteger10 (list-ref tt 0))
                      (uinteger10 (list-ref tt 2))))
                  (else
                   (decimal10 tt))))))
           (real10
            (lambda (t)
              (* (sign (list-ref t 0)) (ureal10 (list-ref t 1)))))
           (opt
            (lambda (op t default)
              (if (null? t)
                  default
                  (op (list-ref t 0)))))
           (complex10
            (lambda (t)
              (let ((tt (cadr t)))
                (case (car t)
                  ((0)
                   (real10 tt))
                  ((1)
                   (make-polar (real10 (list-ref tt 0))
                               (real10 (list-ref tt 2))))
                  (else
                   (make-rectangular
                    (opt real10 (list-ref tt 0) 0)
                    (* (if (char=? (list-ref tt 1) #\+) 1 -1)
                       (opt ureal10 (list-ref tt 2) 1)))))))))
           (num10
            (lambda (t)
              ((prefix10 (list-ref t 0))
               (complex10 (list-ref t 1)))))
           (number
            (lambda (t)
              (num10 t))))
      (number t))))
```

Figure 7. Implementation of a helper function for the lexical analysis of numbers.

one cannot get rid of, even when parse tree construction is almost never used.

9. Discussion

As far as we know, the original technique is the only one that makes automatically generated lexical analyzers able to build parse trees for lexemes using only finite-state tools and this work is the only implementation of it.

Generated lexical analyzers always give access to the matched lexemes. It is essential for the production of tokens in lexical analysis. To also have access to information that is automatically extracted from the lexemes is a useful feature. However, when such a feature is provided, it is typically limited to the ability to extract sub-lexemes that correspond to tagged (e.g. using `\(` and `\)`) sub-expressions of the regular expression that matches the lexeme. Techniquely, for efficiency reasons, it is the *position* and the *length* of the sub-lexemes that get extracted. The IEEE standard 1003.1 describes, among other things, which sub-lexemes must be extracted [7]. Ville Laurikari presents an efficient technique to extract sub-lexemes in a way that complies with the standard [11]. In our opinion, extraction by tagging is too restrictive. The main problem is that, when a tagged sub-expression lies inside of a repetition operators (or inside of what is sometimes called a *non-linear* context) and this sub-expression matches many different parts of a given lexeme, only one of the sub-lexemes is reported. So extraction by tagging starts to become ineffective exactly in the situations where the difficulty or the sophistication of the extraction would make automated extraction most interesting.

Since the conventional way of producing parse trees consists in using a syntactic analyzer based on context-free grammar technology, one might consider using just that to build parse trees for his lexemes. For instance, one could identify lexemes using a DFA and then submit the lexemes to a subordinate syntactic analyzer to build parse trees. Alternatively, one could abandon finite-state technology completely and directly use a *scanner-less* syntactic analyzer. However, both options suffer from the fact that analyzers based on context-free grammars are much slower than those based on finite-state automata. Moreover, an ambiguous regular expression would be translated into an ambiguous context-free grammar. Our technique handles ambiguous expressions without problem but most parsing technology cannot handle ambiguous grammars. Of course, there exist parsing techniques that can handle ambiguous grammars, such as Generalized LR Parsing [10, 15], the Earley algorithm [5], or the CYK algorithm [8, 17, 2], but these exhibit worse than linear time complexity for most or all ambiguous grammars. Finally, it is possible to translate any regular expression into an unambiguous left- or right-linear grammar [6]. However, the resulting grammar would be completely distorted and would lead to parse trees that have no connection to the parse trees for lexemes that we introduced here.

10. Future work

- We intend to complete the integration of automatic parse tree construction into SILex and to clean up the whole implementation.
- Parse tree construction could be made faster. In particular, when the DFA is represented as Scheme code, the functions that build the trees ought to be specialized code generated from the information contained in the three tables.
- The penalty that is strictly due to the additional instrumentation (i.e. when no parse trees are requested) ought to be reduced. A way to improve the situation consists in marking the deterministic states that may reach an accepting state that corresponds to a rule that requests the construction of a parse tree. Then, for states that are *not* marked, the instrumentation that records the path in the DFA could be omitted.
- All tables generated by SILex ought to be compacted but the one for f, in particular, really needs it. Recall that f takes a three-dimensional input and returns a variable-length output (a pair that contains a sequence of commands).
- Some or all of the following regular operators could be added to SILex: the difference (denoted by, say, $r_1 - r_2$), the complement ($\overline{r}$), and the intersection ($r_1 \& r_2$). Note that, in the case of the complement operator, there would be no meaningful notion of a parse tree for a lexeme that matches $\overline{r}$. In the case of the difference $r_1 - r_2$, the parse trees for a matching lexeme w would be the demonstration that r_1 generates w. Finally, in the case of the intersection, for efficiency reasons, only one of the sub-expressions should be chosen to be the one that dictates the shape of the parse trees.
- The parse trees act as (too) detailed demonstrations. Almost always, they will be either transformed into a more convenient structure, possibly with unnecessary details dropped, or completely consumed to become non-structural information. In other words, they typically are transient data. Consequently, it means that only their informational contents were important and that they have been built as concrete data structures to no purpose. In such a situation, deforestation techniques [16] could be used so that the consumer of a parse tree could virtually traverse it even as it is virtually built, making the actual construction unnecessary.

11. Conclusion

This paper presented the adaptation and the implementation of the automated construction of parse tree for lexemes. The technique that has been adapted was originally presented in 2000 by Dubé and Feeley. It has been implemented and integrated in SILex, a lexical analyzer generator for Scheme.

The adaptation was a simple step as it consisted only in modifying the automaton construction rules of the original technique so that the larger set of regular operators of SILex was handled and so that the way the parse trees are built match the right-to-left direction in which lists are built in Scheme.

The implementation was a much more complicated task. Fortunately, SILex, like the original technique, is based on the construction of non-deterministic automata that get converted into deterministic ones. Still, most parts of the generator had to be modified more or less deeply and some extensions also had to be made to the analysis-time module of the tool. Modifications have been done to the representation of the regular expressions, to the way the non-deterministic automata are built and represented, to the conversion of the automata into deterministic ones, to the printing of SILex's tables, to the generation of Scheme code that forms parts of the lexical analyzers, to the algorithm that recognize lexemes, and to the (previously inexistent) construction of the parse trees.

The new version of SILex does work, experiments could be run, but the implementation is still somehow disorganized. The construction of parse trees is a pretty costly operation compared to the normal functioning of a deterministic automaton-based lexical analyzer and, indeed, empirical measurements show that its intensive use roughly triples the execution time of an analyzer.

Acknowledgments

We wish to thank the anonymous referees whose comments really helped to improve this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[3] D. Dubé. Scheme Implementation of Lex, 2001. `http://www.iro.umontreal.ca/~dube/silex.tar.gz`.

[4] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, feb 1970.

[6] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesly Publishing Company, 1979.

[7] IEEE std 1003.1, 2004 Edition.

[8] T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, USA, 1965.

[9] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, aug 1998.

[10] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269, London, UK, 1974. Springer-Verlag.

[11] Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 181–187, sep 2000.

[12] M. E. Lesk. Lex—a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.

[13] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, 2nd edition, 1992.

[14] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[15] M. Tomita. Efficient parsing for natural languages. *A Fast Algorithm for Practical Systems*, 1986.

[16] P. L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[17] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

Rapid Case Dispatch in Scheme

William D Clinger
Northeastern University
will@ccs.neu.edu

Abstract

The `case` expressions of Scheme can and should be implemented efficiently. A three-level dispatch performs well, even when dispatching on symbols, and scales to large `case` expressions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization

General Terms Algorithms, Languages, Performance

Keywords case expressions, symbols, Scheme

1. Introduction

Programming languages should be implemented not by piling hack upon hack, but by removing the inefficiencies and restrictions that make additional hacks appear necessary.

The `case` expressions of Scheme are a convenient syntax for rapid selection between actions determined by a computed value that is expected to lie within a known finite set of symbols, numbers, characters, and booleans [5].

Although Scheme's `case` expressions are fast by design, too many systems still implement them inefficiently. These inefficient implementations have led some programmers to write contorted and inefficient code for case dispatch when `case` expressions would have been more natural *and* more efficient.

In particular, some Scheme programmers believe the evaluation of a `case` expression requires time proportional to the number of literals mentioned in its clauses. Others believe the efficiency of multi-way case dispatch on characters depends upon the size of the character set. Some understand that multi-way case dispatch on numbers and characters is efficient, but believe that multi-way case dispatch on symbols is inherently inefficient. These incorrect beliefs have led some programmers to eschew the use of symbols as enumerated values, to fear Unicode, or to avoid `case` expressions altogether.

The contributions of this paper are:

1. To show that Scheme's `case` expressions are efficient when implemented properly.
2. To describe an efficient triple-dispatch technique for implementing general case dispatch.

The techniques used to implement fast case dispatch in languages like Pascal, C, and Java are well-known, so the primary focus of this paper is on more Scheme-specific issues: fast dispatch for symbols and for dispatch on values of mixed types.

2. Implementation

This section describes the implementation of `case` expressions in Larceny v0.92, which uses the Twobit compiler [1, 4].

The basic idea can be seen in figure 1:

1. Dispatch on the type.
2. Use some type-specific dispatch to map the value to the index of its associated clause.
3. Use binary search on the clause index to select the expressions to be evaluated for that clause.

What remains to be explained are the details. Following Orbit's example [7], Twobit's first pass macro-expands `case` expressions into more primitive expressions, as described in R5RS 7.3 [5].

When control optimization is enabled, Twobit's second pass recognizes `if` expressions whose test is a call

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

```
(let ((n (cond ((char? var0)
                <dispatch-on-char>)
               ((symbol? var0)
                <dispatch-on-symbol>)
               ; miscellaneous constants
               ((eq? var0 '#f) ...)
               ...
               ((fixnum? var0)
                <dispatch-on-fixnum>)
               (else 0))))
  <dispatch-on-n>)
```

Figure 1. General form of triple dispatch

to `eq?`, `eqv?`, `memq`, or `memv` whose first argument is a variable and whose second argument is a literal constant. When such an expression is found, Twobit looks for nested `if` expressions of the same form whose test compares the same variable against one or more literal constants. Twobit analyzes these nested `if` expressions to reconstruct the equivalent of a set of `case` clauses, each consisting of a set of constants paired with the expressions to be evaluated if the variable's value is one of those constants. This analysis removes duplicate constants, so the sets of constants are disjoint.

Twobit then decides between one of two strategies:

- brute-force sequential search
- the triple dispatch of figure 1

Sequential search is used if the total number of constants is less than some threshold, typically 12, for which benchmarks have shown the triple-dispatch technique to be faster than a simple sequential search.

If Twobit decides to use triple dispatch, then it numbers the clauses sequentially (reserving 0 for the `else` clause, if any) and generates code of the form shown in figure 1. If there are no miscellaneous constants, then the corresponding `cond` clauses will not appear. If there are no character constants, then the character clause is unnecessary, and similarly for the symbol and fixnum clauses.

(A *fixnum* is a small exact integer. Twobit's idea of the fixnum range may be smaller than the fixnum range that is actually defined by some of Larceny's back ends, so Twobit may misclassify a large fixnum as a miscellaneous constant. That misclassification is safe because the miscellaneous constants come before the `fixnum?` test in figure 1.)

```
(lambda (x)
  (case x
   ((#\a #\e #\i #\o #\u #\A #\E #\I #\O #\U
     a e i o u)
    (f-vowel x))
   ((#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m
     #\n #\p #\q #\r #\s #\t #\v #\w #\x #\y #\z
     #\B #\C #\D #\F #\G #\H #\J #\K #\L #\M
     #\N #\P #\Q #\R #\S #\T #\V #\W #\X #\Y #\Z
     b c d f g h j k l m n p q r s t v w x y z)
    (f-consonant x))
   (else
    (f-other x))))
```

Figure 2. Example: source code

The three type-specific dispatches are independent, and can be implemented in completely different ways.

To map a fixnum to a clause index, Twobit chooses one of these techniques:

- sequential search
- binary search
- table lookup

Sequential search is best when there are only a few fixnum constants, with gaps between them. The cost of a binary search depends on the number of intervals, not on the number of constants; for example, the cost of testing for membership in $[1, 127]$ is the same as the cost of testing for membership in $[81, 82]$. The choice between binary search and table lookup is made on the basis of code size: a binary search costs about 5 machine instructions per interval, while a table lookup costs about $hi - lo$ words, where *lo* and *hi* are the least and greatest fixnums to be recognized. Binary search followed by table lookup would be an excellent general strategy, but Twobit does not yet combine binary search with table lookup.

To map a character to a clause index, Twobit converts the character to a fixnum and performs a fixnum dispatch.

To map a symbol to a clause index, Twobit can use either sequential search or a hash lookup. In Larceny, every symbol's hash code is computed when the symbol is created and is stored explicitly as part of the symbol structure, so hashing on a symbol is very fast. Twobit uses a closed hash table, represented by a vector of symbols (or `#f`) alternating with the corresponding clause index (or 0 for the `else` clause). As this vector is

```
(lambda (x)
  (let* ((temp x)
         (n (if (char? temp)
                (let ((cp (char->integer:chr temp)))
                  (if (<:fix:fix cp 65)
                      0
                      (if (<:fix:fix cp 124)
                          (vector-ref:trusted
                            '#(1 2 2 2 1 2 2 2 1 2 2 2 2 2 1 2
                               2 2 2 2 1 2 2 2 2 2 0 0 0 0 0 0
                               1 2 2 2 1 2 2 2 1 2 2 2 2 2 1 2
                               2 2 2 2 1 2 2 2 2 2 0)
                            (-:idx:idx cp 65))
                          0)))
                (if (symbol? temp)
                    (let ((symtable
                           '#(#f 0 #f 0 #f 0 #f 0 w 2 x 2 y 2 z 2
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0 #f 0
                              c 2 d 2 e 1 f 2 #f 0 #f 0 a 1 b 2
                              k 2 l 2 m 2 n 2 g 2 h 2 i 1 j 2
                              s 2 t 2 u 1 v 2 o 1 p 2 q 2 r 2))
                          (i (fixnum-arithmetic-shift-left:fix:fix
                              (fixnum-and:fix:fix 63 (symbol-hash:trusted temp))
                              1)))
                      (if (eq? temp (vector-ref:trusted symtable i))
                          (vector-ref:trusted symtable (+:idx:idx i 1))
                          0))
                    0))))
    (if (<:fix:fix n 1)
        (f-other x)
        (if (<:fix:fix n 2)
            (f-vowel x)
            (f-consonant x)))))
```

Figure 3. Example: partially optimized intermediate code

generated, Twobit computes the maximum distance between the vector index computed from a symbol's hash code and the vector index at which the symbol is actually found. This bound on the closed hash search allows Twobit to generate straight-line code, without loops.

All of the fixnum, character, symbol, and vector operations that implement these strategies will operate on values that are known to be of the correct type and in range, so most of those operations will compile into a single machine instruction.

Figures 2 and 3 show the complete code for an artificial example. (For this example, all of the symbols are found at the vector index computed from their hash code, so no further search is necessary. The intermediate code has been edited to improve its readability.)

Twobit's optimization of `case` expressions could have been performed by implementing `case` as a low-level macro. This would be slightly less effective than what Twobit actually does, because Twobit will optimize nested `if` expressions that are equivalent to a `case` expression, even if no `case` expression was present in the original source code. The macro approach may nonetheless be the easiest way to add efficient case dispatch to simple compilers or interpreters.

3. Survey

An incomplete survey of about 140,000 lines of Scheme code distributed with Larceny v0.92 located about 330 `case` expressions [4]. Of the 180 that were examined in detail, the largest and most performance-critical were found within various assemblers and peephole optimizers, written by at least three different programmers. The largest `case` expression is part of Common Larceny's new in-memory code generator (for which the fashionable term would be "JIT compiler"), and translates symbolic names of IL instructions to the canonical strings expected by Microsoft's `System.Reflection.Emit` namespace. This `case` expression contains 217 clauses with 363 symbols. The next largest contains 17 clauses with 102 symbols. Four `case` expressions contain 32 to 67 fixnum literals, and another dozen or so contain 16 to 31 symbols.

Only seven of the 180 `case` expressions contain literals of mixed type. One is the 217-clause monster, which contains 214 lists as literals in addition to its 363 symbols, but those list literals are useless and derive from an otherwise benign bug in a local macro; the 363 symbols should have been the only literals. (Had the list literals slowed this case dispatch, loading a source file into Common Larceny would be even slower than it is.) The mixed types in three other `case` expressions were caused by that same bug. The three purposeful examples of mixed-type dispatch contain 7, 10, or 11 literals, mixing symbols with booleans or fixnums, and their performance is unimportant. Mixed-case dispatch appears to be more common in the less performance-critical code whose case expressions were not examined in detail.

4. Benchmarks

Source code for the benchmarks described in this section is available online [2].

A six-part `case` micro-benchmark was written to test the performance of case dispatch on fixnums and on symbols, for `case` expressions with 10, 100, or 1000 clauses that match one fixnum or symbol each. Figure 4 shows the 10-clause `case` expression for symbols, from which the other five parts of the micro-benchmark can be inferred. Each of the six parts performs one million case dispatches, so any differences in timing between the six parts must be attributed to the number of clauses in each case dispatch, and to the difference between dispatching on a fixnum and dispatching on a symbol.

```
(define (s10 x)
  (define (f x sum)
    (case x
      ((one) (f 'two (- sum 1)))
      ((two) (f 'three (+ sum 2)))
      ((three) (f 'four (- sum 3)))
      ((four) (f 'five (+ sum 4)))
      ((five) (f 'six (- sum 5)))
      ((six) (f 'seven (+ sum 6)))
      ((seven) (f 'eight (- sum 7)))
      ((eight) (f 'nine (+ sum 8)))
      ((nine) (f 'onezero (- sum 9)))
      ((onezero) (f 'oneone (+ sum 10)))
      (else (+ sum 9))))
  (f x 0))
```

Figure 4. One part of the `case` micro-benchmarks

The `monster` micro-benchmark is a mixed-type case dispatch that uses the 217-clause, 577-literal `case` expression of Common Larceny v0.92 to translate one million symbols to strings. (That many translations might actually occur when a moderately large program is loaded into Common Larceny.)

A set of four benchmarks was written to measure performance of Scheme systems on components of a realistic parsing task [2]. The `parsing` benchmark reads a file of Scheme code, converts it to a string, and then parses that string repeatedly, creating the same data structures the `read` procedure would create. The timed portion of the `parsing` benchmark begins after the input file has been read into a string, and does not include any i/o. The `lexing` and `casing` benchmarks are simplifications of the `parsing` benchmark, and measure the time spent in lexical analysis and in case dispatch, respectively. (The `lexing` benchmark computes the same sequence of lexical tokens that are computed by the `parsing` benchmark, but does not perform any other parsing. The main differences between the `lexing` benchmark and the `casing` benchmark are that the `casing` benchmark does not copy the characters of each token to a token buffer and does not keep track of source code locations. The `casing` benchmark still includes all other string operations that are performed on the input string during lexical analysis, so it is not a pure case dispatch benchmark.) The `reading` benchmark performs the same task as the `parsing` benchmark, but uses the built-in `read` procedure to read from a string port (SRFI 6 [3]). The main purpose of the `reading` benchmark is to show that the

parsing benchmark's computer-generated lexical analyzer and parser are not outrageously inefficient.

Both the state machine of the lexical analyzer and the recursive descent parser were generated by the author's LexGen and ParseGen, which can generate lexical analyzers and parsers written in Scheme, Java, or C [2]. This made it fairly easy to translate the parsing benchmark into Java. As it was not obvious whether the strings of the Scheme benchmark should be translated into arrays of char or into instances of the StringBuilder class, two versions of the Java code were written; a third version, just for grins, uses the thread-safe StringBuffer class.

The timings reported in the next section for the casing, lexing, parsing, and reading benchmarks were obtained by casing, lexing, parsing, or reading the nboyer benchmark one thousand times [2].

5. Benchmark Results

Tables 1 and 2 show execution times for the benchmarks, in seconds, as measured for several implementations on an otherwise unloaded SunBlade 1500 (64-bit, 1.5-GHz UltraSPARC IIIi). Most of the timings represent elapsed time, but a few of the slower timings represent CPU time. For the compiled systems and the fastest interpreters, the timings were obtained by averaging at least three runs. For the slower interpreters, the reported timing is for a single run.

For the two largest case micro-benchmarks, three of the Scheme compilers generated C code that was too large or complex for gcc to handle.

From table 1, it appears that compilers C and D use sequential search for all case expressions. Compilers B, E, and F generate efficient code when dispatching on fixnums, but appear to use sequential search for symbols.

Compiler A (Larceny v0.92) has the best overall performance on the micro-benchmarks, and Compiler B (Larceny v0.91) is next best. The difference between them is that Larceny v0.92 implements case expressions as described in this paper.

Table 2 shows that, for the parsing benchmark, most of these implementations of Scheme spend roughly half their time in case dispatch. The two that spend the least time in case dispatch, compilers F and B, perform well on the fixnum case micro-benchmarks and appear to be doing well on the parsing benchmark's character dispatch also. Compiler C's performance may mean sequential search is fast enough for this benchmark, or it may mean that compiler C recognizes case clauses that match sets of consecutive characters (such as #\a through #\z, #\A through #\Z, and #\0 through #\9) and tests for them using a range check instead of testing individually for each character.

The difference between Larceny v0.92 and v0.91 (compilers A and B) does not matter for the parsing benchmark, because v0.91 was already generating efficient code for case dispatch on characters.

6. Related Work

Compilers for mainstream languages typically implement case/switch statements using sequential search, binary search, or jump tables [6].

A binary search usually concludes with a jump to code for the selected case. In the subset of Scheme that serves as Twobit's main intermediate language, jumps are best implemented as tail calls. Those calls would interfere with many of Twobit's intraprocedural optimizations, so Twobit does not use a single-level binary search.

Jump tables are hard to express in portable Scheme without using case expressions, which are not part of Twobit's intermediate language. Adding even a restricted form of case expressions to Twobit's intermediate language is unattractive, because it would complicate most of Twobit's other optimizations.

Jump tables can be implemented portably using a vector of closures, but it would cost too much to create those closures and to store them into a vector every time the scope containing a case expression is entered. A vector of lambda-lifted closures could be created once and for all, but would entail the costs of passing extra arguments and of making an indirect jump. With either form of jump table, calling a closure that cannot be identified at compile time would interfere with many of Twobit's intraprocedural optimizations.

The Orbit compiler demonstrated that it is practical to macro-expand case expressions into if expressions, and for control optimization to recognize and to generate efficient code from those if expressions [7].

Acknowledgments

The author is grateful to the anonymous reviewers, not only for remarks that improved this paper, but also for suggestions that improved the implementation of case expressions in Larceny v0.92 [4].

	`case`						`monster`
	10 literals		100 literals		1000 literals		577 literals
	fixnum	symbol	fixnum	symbol	fixnum	symbol	mixed
Compiler A	.04	.05	.04	.08	.11	.13	.16
Compiler B	.04	.05	.07	.21	.14	3.94	3.04
Compiler C	.04	.04	.18	.17	3.80	4.61	8.33
Compiler D	.09	.09	.24	.22	—	—	15.94
Compiler E	.06	.12	.02	.50	—	—	15.11
Compiler F	.05	.70	.04	6.03	—	—	26.95
Interpreter G	1.52	1.30	10.00	7.80	96.81	78.03	26.21
Interpreter H	1.79	1.76	10.65	10.91	115.52	119.67	
Interpreter I	3.48	3.48	15.62	15.38	188.12	186.38	33.50
Interpreter J	6.00	6.33	20.99	21.63	193.17	196.21	60.26
Interpreter K	5.00	5.00	21.00	24.00	211.00	256.00	59.00
Interpreter L	5.36	5.38	29.09	28.30	280.22	289.58	147.43
Interpreter M	6.12	4.48	49.48	30.42	447.08	301.53	338.78
Interpreter N	13.82	13.88	77.68	78.18	757.16	776.75	459.51

Table 1. Timings in seconds for the `case` and `monster` micro-benchmarks

	`casing`	`lexing`	`parsing`	`reading`
HotSpot (array of `char`)			11.05	
HotSpot (`StringBuilder`)			12.21	
Compiler C	7.36	10.67	13.27	2.23
Compiler F	2.83	5.39	14.48	2.60
Compiler B	6.93	12.84	21.17	14.67
HotSpot (`StringBuffer`)			24.95	
Compiler D	13.53	22.65	27.20	17.78
Compiler E	45.67	63.88	84.46	72.53
Interpreter G	79.93	108.44	128.95	13.88
Interpreter H	82.80	116.12	214.82	18.98
Interpreter I	180.64	237.37	297.96	
Interpreter L	257.13	383.77	432.13	
Interpreter J	436.19	566.83	645.31	
Interpreter M	479.36	589.17	701.70	
Interpreter K	468.00	628.00	745.00	
Interpreter N	1341.93	1572.89	1793.64	

Table 2. Timings in seconds for `parsing` and related benchmarks

References

[1] Clinger, William D, and Hansen, Lars Thomas. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, 1994, pages 128–139.

[2] Clinger, William D. Source code for the benchmarks described in this paper are online at `www.ccs.neu.edu/home/will/Research/SW2006/`

[3] Clinger, William D. SRFI 6: Basic String Ports. `http://srfi.schemers.org/srfi-6/`.

[4] Clinger, William D., et cetera. The Larceny Project. `http://www.ccs.neu.edu/home/will/Larceny/`

[5] Kelsey, Richard, Clinger, William, and Rees, Jonathan (editors). Revised^5 *report on the algorithmic language Scheme.* ACM SIGPLAN Notices 33(9), September 1998, pages 26–76.

[6] Fischer, Charles N., and LeBlanc Jr, Richard J. *Crafting a Compiler with C*. Benjamin/Cummings, 1991.

[7] Kranz, David, Adams, Norman, Kelsey, Richard, Rees, Jonathan, Hudak, Paul, and Philbin, James. ORBIT: an optimizing compiler for Scheme. In *Proc. 1986 SIGPLAN Symposium on Compiler Construction*, 1986, pages 219–233.

A. Notes on Benchmarking

The benchmarked systems:

HotSpot is the Java HotSpot(TM) Client VM of Sun Microsystems (build 1.5.0_01-b08, mixed mode, sharing).

A is Larceny v0.92.

B is Larceny v0.91.

C is Chez Scheme v6.1.

D is Gambit 4.0b17.

E is Chicken Version 1, Build 89.

F is Bigloo 2.7a.

G is MzScheme v352.

H is MzScheme v301.

I is the Larceny v0.92 interpreter.

J is the Gambit 4.0b17 interpreter.

K is the Bigloo 2.7a interpreter.

L is the MIT Scheme 7.3.1 interpreter.

M is the Scheme 48 1.3 interpreter.

N is the Chicken 1,89 interpreter.

Except for MzScheme, the interpreters were benchmarked with no declarations and with the default settings. The compilers and MzScheme were benchmarked as in Chez Scheme's `(optimize-level 2)`: safe code, generic arithmetic, inlining the usual procedures. Specifically:

A was compiled with `(compiler-switches 'fast-safe)` and `(benchmark-mode #f)`.

B was compiled the same as A.

C was compiled with `(optimize-level 2)`.

D was compiled with `-prelude "(declare (extended-bindings))" -cc-options "-O2" -dynamic`.

E was compiled with `-no-trace -optimize-level 2 -block -lambda-lift`.

F was compiled as a module with `-O6 -copt -O2`.

G was interpreted as a module.

H was interpreted the same as G.

Experiences with Scheme in an Electro-Optics Laboratory

Richard A. Cleis
Air Force Research Laboratory

Keith B. Wilson
Air Force Research Laboratory

Abstract

The Starfire Optical Range is an Air Force Research Laboratory engaged in Atmospheric Research near Albuquerque, New Mexico. Since the late 1980's it has developed numerous telescope systems and auxiliary devices. Nearly all are controlled by C programs that became difficult to manage due to the large number of configurations required to support the experiments. To alleviate the problem, Scheme has been introduced in at least six distinct ways. This paper describes the uses of Scheme, emerging programming techniques, and general experiences of the past several years.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming — Scheme
; D.2.3 [*Programming Techniques*]: Coding Tools and Techniques — DrScheme, MzScheme
; J.2 [*Computer Applications*]: Physical Sciences and Engineering — aerospace, astronomy, engineering

General Terms Algorithms, Design

Keywords embedding, scripting, extension language, motion control, servo, communication, remote development

1. Introduction

Laboratory Mission The primary mission of the Starfire Optical Range (SOR) is to develop and demonstrate optical wavefront control technologies. In other words, it builds systems that remove the distortion caused by air turbulence when light propagates through the atmosphere. [1] The site also supports field experiments by others within the research community.

Experiments are conducted on five permanent precision telescope systems. Many experiments involve multiple telescopes and all telescopes are used for multiple experiments. Some telescopes have components that must move in concert with the main gimbals and most have at least one tracking system. The variety of systems is significant; the largest telescope has a 3.5 meter diameter primary mirror and weighs nearly 150 tons, while the smallest has an aperture of about a quarter meter and weighs about one half ton.

Tracking requirements include celestial objects, artificial satellites, aircraft, balloons, lunar retro-reflectors, interplanetary spacecraft, Space Shuttles returning from space, Leonids meteor trails, terrestrial vehicles, and diagnostic sites downrange of the SOR.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

Laboratory Environment The following lists highlight, from the perspective of developing and operating telescope systems, the nature of the laboratory and strategic improvements that could increase the effectiveness of laboratory software.

- Five telescope systems are run with about a dozen computers.
- As many as three telescopes are needed for a single experiment.
- Two developers do most telescope related programming.
- About eight developers are needed to program all systems.
- Most telescopes are used for several long term experiments.
- At least several operators are needed for most experiments.
- Operators have only casual knowledge of the technology.
- Principle Investigators normally have only casual knowledge of the software.
- New experiments sometimes require new computations and configurations.
- Debugging is sometimes only possible during experiments.
- Subsytems need to communicate during experiments.
- Extensive legacy C software is essential to real-time operations.
- Planning requires significant computations.
- Software developers must consider safety.

At least several of us believe that software could be significantly more effective if the following issues were addressed:

- A non-expert programmer needs to be able to maintain the computational and control systems. Currently, only one developer can maintain or improve them.
- Software is needed to reliably repeat operations as much as several years apart. Currently, if specific operations are to be repeated, the original crew is normally required; this is often difficult for a variety of reasons.
- Principle Investigators need to be provided with clear scripting interfaces for their experiment, even if they only use them to understand and record the procedures.
- Non-expert programmers need to be able to program the creations of the expert computer engineers; most programs can only be maintained by the original engineer.
- Maintenance and performance verification needs to be automated so that they are accomplished more often and trends can be identified.
- Experiments need to be more automated. Most events are sequenced by operators pushing virtual buttons on user interfaces.

Why Scheme? We chose Scheme to address the above issues for a number of reasons. Its elegant syntax appropriately matches the

modest programming requirements, and macros are used to provide simpler syntaxes for scripting. S-expressions are suitable for ethernet communications between dozens of our platforms, including those hosting scripting environments. The same s-expressions may be created and read by C++ and Java programs; this allows Scheme to be used in the lab without requiring all programmers to adopt it. Garbage collection, the absence of assignment statements, and the ability to redefine functions while programs are running are capabilities essential to the rapid development that is occasionally needed while experiments are in progress.

As a Functional Programming Language, Scheme is suitable even in scripting environments that are mainly used to sequence the movement of equipment. Beneath the procedural scripting, functional programming elegantly access values on remote systems, apply filters, and delivers the results to other remote systems. Scheme is also appropriate for eventually extending our C language planning and scheduling software; functional programming in Scheme could more sensibly manipulate lists which are normally built from the results of functions related to celestial and orbital mechanics.

MzScheme, in particular, is embeddable in C and extendable by C. Both techniques are used to access hardware. MzScheme is also a reliable environment for subsystems that run continuously.

Aside from the useful tools that are provided, we chose PLT Scheme because it is supported by numerous teaching aids: web tutorials, textbooks, and an active online forum. The efforts expressed in this paper represent a shift in the implementation of telescope system computers at the SOR; the changes would reverse if learning materials were not available to the programmers who choose to see the value of Scheme.

2. Six Uses for Scheme

The SOR is an electro-optics laboratory that depends on numerous devices which were nearly all programmed in C over the past two decades; C is the natural choice because most programs require access to the hardware bus. Experiments require the specific configuration of up to ten telescope subsystems, a problem which becomes more difficult to manage as the laboratory grows, more devices are built, and more experiments are attempted. To alleviate these problems, extension languages have been implemented in many of the motion control devices such as the telescope gimbals and large optical components. These telescope gimbals consist of precision bearing structures, accurate position transducers, and embedded motors which form closed-loop servos that precisely move the telescopes.

We are using Scheme in six different ways: embedding in C, Scheme motion controllers, C-language gimbals servos interfaced with s-expression parsers, the remote command and status paradigm for all telescope systems, scripting environments for experiments, and the remote development environment for gimbals servos. Highlights are shown in this section.

Embedding in C We embedded MzScheme in the site's ubiquitous motion control application, the Starfire Optical Range Acquisition and Pointing System (SORAPS). This C application had become unwieldy after numerous capabilities were gradually added, since 1987, to accommodate all experiments conducted on the principle telescopes. We also embedded MzScheme in several programs required for the operation of subsystems. Configuring and controlling these programs is significantly more efficient using Scheme because behavior can be changed without rebuilding C.

Motion Control Systems We developed Scheme motion control systems for two optical rotators and the 3-axis secondary mirror controller on the largest telescope. Using DrScheme, we prototyped these relatively low bandwidth controllers then installed the final program on a single board computer that continuously runs MzScheme. We could have used one of the commercial servo development tools acquired for other projects, but they impose restrictions on hardware and software. MzScheme enabled us to build unique controllers with open communications; they can be ported to a variety of ordinary hardware capable of running MzScheme.

Interface to Servos for Large Gimbals The gimbals servos for the three largest telescopes are C language applications linked to the Small Fast S-expression library. [2] We extended the library to provide rudimentary single-depth evaluation of s-expressions that are either received via ethernet or read from a configuration file. This approach allows the servos to be configured and accessed as if they had Scheme embedded, maintaining consistency with the Scheme systems yet requiring only a very light weight library.

Paradigm for Remote Commands and Status S-expressions are the only messages used for remote commands and status of the telescope systems. Other devices (e.g. optical image stabilizers and telemetry servers) also communicate to the system with Scheme commands, even if formed with a printf statement in C. The typical "bit-speak" often found in hardware interface documents is replaced with elegant and self-documenting s-expressions.

Scripting Environment for Experiments We use DrScheme as a scripting environment to automate the telescope and subsystems during experiments. The scripts consists of macros that access support functions which maintain communications with remote systems and control their activity. DrScheme enables our most complex telescope experiments to be conducted using simple keywords that may be manipulated without interrupting operations.

Remote Development Environment We wrote tools in DrScheme for remotely running and testing the gimbals servos of the three largest telescopes. The "viewport graphics" displayed performance while we used the REPL and threads to move the gimbals and optimize properties contained in the servo. After developing the tools for a new controller for the largest telescope, we used them to complete similar controllers on two other telescopes. Only a few days were required for each of the second two because DrScheme had enabled us to easily write and reuse tools specific to large telescopes.

3. Programs and Tools

The first author wrote most of the following software, the second author has been writing the programs for the newest subsystems and has been configuring the diskless linux platforms that host most of the controllers.

Major Programs The major programs are SORAPS and NMCS. Each of five telescopes needs to be connected via ethernet to one of many installations of SORAPS, a full featured C program that handles operations, planning, and real-time computation. MzScheme was embedded to aid configuration and communication.

NMCS is a gimbals servo written in C and linked to the Small Fast S-Expression Library. The gimbals for the three largest telescopes require a separate NMCS; each runs continuously on a single board computer. SORAPS and NMCS are linked via ethernet and communicate exclusively with s-expressions. The remaining two gimbals are controlled via serial ports connected to SORAPS.

Minor Programs Embedded MzScheme programs are used to control a secondary telescope mirror and bridge new s-expressions to legacy subsystems that require proprietary communication formats. These programs communicate to both SORAPS and the scripting environment for purposes such as dome control, optical configuration and alignment, and telescope focus.

Two optical rotators are controlled by programs we wrote for MzScheme, using a few extensions to access the hardware. These

rotators communicate with SORAPS to report their position and receive commands.

Applications we created in DrScheme include simulators for gimbals, telemetry, an image-stabilizer, a rotator, and a communication bridge.

Developmental Software We developed a suite of functions and a module to script the telescope systems for complex experiments. We also developed servo development tools that were used to optimize NMCS for each of the three telescope gimbals.

Libraries, Extensions, and Development Tools Programs that embed MzScheme are linked to libmzgc.a and libmzscheme.a. SORAPS and all of its supporting Scheme files are contained in a volume which can be mounted by any OS X platform. Because the libraries are linked to the application, platforms can run SORAPS even if no Scheme environments are installed.

NMCS programs are linked to libsexp.a, built from the source files of the Small Fast S-Expressions project. Two telescopes and one rotator are connected via serial ports implemented in a Scheme extension that we wrote. For all of the development described in this paper, we used DrScheme, KDevelop, and Xcode.

4. Using Scheme to Script a Telescope System

We developed Scheme functions and scripts for a long term series of experiments that began in early 2006. The scripts are designed to be readable by non-programmers and modifiable by personnel who understand the operations. They serve as an executable operational summary as well as an essential tool for operations. A principle script is used to guarantee operations; it assumes no state and performs all known initializations. Other scripts are used to suspend and resume the principle script without performing initializations that would be disruptive.

The scripts are supported by a suite of functions that communicate with the subsystems and perform experiment-specific computations. A module of macros defines key-phrases that can be sequenced by a macro that defines each script.

4.1 The Principle Script

The system needs to point to a moving balloon-borne platform containing scientific instruments. Telemetry messages are requested, filtered, and routed to SORAPS which computes dynamic vectors for the telescope, the dome, the elevation window contained by the dome, and the optical rotator. All subsystems are set into motion, then the script waits until they mechanically converge on the platform. The script also positions several mirrors in the optical path and sets the position of the mirror used for focus. A macro, `define-sor-script`, creates a script that is named with the first parameter:

```
(define-sor-script find-platform
 prepare-subsystem-connections
 verify-subsystem-connections
 prepare-soraps-for-wgs84-propagation
 feed-telemetry-to-soraps
 tell-subsystems-to-follow-soraps
 wait-for-subsystems-to-converge)
```

`Define-sor-script` wraps the script elements in an exception handler that will call a function that stops the system if an error occurs. It then executes the sequence in a thread that can be killed by other scripts, such as `stop`.

```
(define-syntax define-sor-script
  (syntax-rules ()
    ((_ proc-name f1 ...)
     (define-syntax proc-name
       (syntax-id-rules
        ()
(_ (begin
     (keep-thread
      (thread
       (lambda ()
         (with-handlers
             ((exn?
               (lambda (exn)
                 (stop-system
                  (exn-message exn)))))
           (f1) ...
           (display-final-message 'proc-name)))))
     (display-initial-message 'proc-name))))))))
```

The module containing `define-sor-script` manages the thread (saved with `keep-thread`) and defines functions that display initial and final messages in the REPL. The script is defined with `syntax-id-rules` to enable operators to type the name, without parentheses, to execute the script. This elementary macro threads the sequence, stops the system when an exception occurs, and provides diagnostics messages; the script writer is merely required to use `define-sor-script`.

`Prepare-subsystem-connections` is a macro that sets the addresses and ports of connections to all subsystems and starts a thread that evaluates incoming messages. It also sends a message to a a common displayer used to show the occurrence of significant events:

```
(define-syntax prepare-subsystem-connections
    (syntax-id-rules
     ()
     (_ (begin (set-senders-to-subsytems!)
               (start:evaluate-network-inputs)
               (send-to-displayer
                "Connections were created.")))))
```

This macro is typical of those in the rest of the script. It contains functions that provide utility to a script writer, and it has alternatives which make it necessary; macros for simulators could have been used instead. Some macros could have been implemented as functions, but we wanted all elements of the scripting environment to be executable without parentheses. This allows the script writer to test individual macros in the REPL.

More complex scripts include search algorithms and the use of automatic tracking systems, but we have not yet used them.

We developed this elementary environment and started using it for all balloon experiments. However, it is merely a first attempt at scripting, a software experiment inside of a physics experiment. We plan to thoroughly investigate other possibilities, such as creating specific languages for the experiments and subsystems.

4.2 Strategy

Writing scripts and their support functions “from the top, down” biased scripts toward what users want, rather than what programmers want to write. To minimize the use of resources, we developed as much software as possible before requiring the operation of any remote subsystems. Skeleton functions in each layer were debugged with the entire program before the skeletons were “filled in”, creating the need for another layer. The lowest layer provided the network connections; they simulated messages from the remote systems which were not yet available.

This minimal executable script served as executable documentation suitable for preparing for the Critical Design Review. Each network simulation also provided an operating specification for the

development of incomplete subsystems. Eventually, we debugged the connections, one at a time, as we replaced each simulator with the corresponding subsystem.

We used an outliner program to specify the scripts. Simple conditionals and looping can be represented in an outline; anything more complicated was not considered for the scripting layer. After many iterations, the top layer of the outlines evolved into simple lists of procedures that were eventually implemented as macros. The second layer was implemented as the top layer of Scheme functions; so no conditionals or looping was required in the script.

The outline formed what is suggestive of "wish lists" in the text *How to Design Programs*. [4] Functions were written to simulate requirements of the wish lists so that a working simulation could be completed before expensive engineering commenced. In the future, we intend to employ such textbook ideas in documentation for writing scripts and support functions.

We developed the software with the intent that it could be maintained and extended in three levels. The highest level, scripting, is usable by anyone who has familiarity with programming. The lowest level functions are to be maintained by the experts who created the optical-mechanical subsystems. The middle level is for programmers who can understand the needs of the script writers and the operation of the systems. We expect that this strategy will help us effectively manage the software for telescope systems and experiments.

5. Details of Scheme Use

5.1 Embedding in C

We embedded MzScheme in C programs for three reasons. First, it serves as the extension language for the legacy C program, SORAPS. Second, it runs Scheme programs that access hardware in C. Third, it provides a bridge between Scheme programs and proprietary network libraries. Embedding Scheme, here, refers to linking MzScheme with C and creating a single Scheme environment when the C program is started. This environment contains definitions of scheme primitives that access functions previously written in C. Functions in the environment are invoked by the evaluation of Scheme files, the evaluation of expressions received via ethernet, or by calls from the C side of the program.

5.1.1 Extension Language for SORAPS

Embedding MzScheme in SORAPS allows other programs to configure it, enables networked operation, and provides a means for other programs to access essential calculations.

Configuring SORAPS Configuring SORAPS is a difficult problem. Individual SORAPS installations control the five largest telescopes, and multiple installations may control each telescope. Also, temporary subsystems (including telescopes) occasionally need access to the calculations. These configurations cannot be handled elegantly with simple configuration files partly because the telescopes have different command and status requirements. Furthermore, many configuration items will likely be moved from legacy files to an on-site database.

The differences between the telescopes are not trivial abstraction issues. The NMCS control systems are a service provided to SORAPS via ethernet, as SORAPS initiates all transactions with little time restriction. A commercial system behaves as a client to SORAPS, pushing status over a serial port in a precisely synchronous manner. Another commercial system is a combination of the two. Two recently replaced systems had no communication protocol at all; they required a hardware connection to their bus. Furthermore, telescope systems often have components that affect the optical perspective of other components in the system. The Scheme environment of SORAPS contains functions to handle these problems so that computations can be modified for affected devices, yet SORAPS doesn't need to be rebuilt or restarted.

Configuration, command, and status functions are contained in Scheme files that may be manipulated in an external environment like DrScheme. Primitives provide a means for setting properties of the system with relatively unrestricted programs instead of fixed-format files that were required by the original C program.

Serial ports are sometimes needed to interface dynamic equipment such as gimbals controllers. Scheme programs, embedded or independent, load a serial extension and interface functions specific to the application. This is an advantage over earlier solutions that depended on configuration-specific libraries; the Scheme files are available and editable on the operations computer, only a text editor is required to change the behavior of the program significantly.

The following fragment represents the kind of function used for configuring fundamental properties of a telescope:

```
(let ((id '("3.5m Telescope" "cetus")
          ; long name, short name (no spaces)
          )
      (loc (list wgs84-semimajor-axis
                 wgs84-inverse-flattening-factor
                 ;any ellipsoid may be used
                   34.9876543 ; deg latitude
                 -106.456789  ; deg longitude
                 1812.34      ; met height
                 6 ; hours local to utc in summmer
                 )))
  (display (set-system-gimbals cetus id loc ))
  (newline))
```

`Set-system-gimbals` is a primitive in C, it returns readable messages that indicate success or failure. "Wgs84..." are numbers that are defined in Scheme, but any reference ellipsoid may be used; on rare occasions, researchers want to use their own.

An example for configuring an evaluator of network messages:

```
(define eval-socket (udp-open-socket))
(define eval-the-socket
        (make-function-to-eval eval-socket 4096))
```

`Make-function-to-eval` was written to support Scheme communications (it is explained later.) Its product, `eval-the-socket`, is either used in a thread or it can be called by scheduling functions invoked by C. Both the behavior of the communications and their implementation are entirely handled in Scheme, a more effective environment than C for maintaining the communications.

Operating SORAPS We installed primitives in SORAPS to allow the selection of computations and the control of its associated telescopes. For telescope movement, a mode primitive is installed to evaluate expressions like:

```
(do-mode telescope-index 'stop)
```

Changing between `stop` and `vect` makes a telescope stop or follow vectors that are supplied by other expressions. Following is the C function needed for the primitive.

```
static Scheme_Object
*do_mode( int nArgs, Scheme_Object* args[] )
{
  enum { SYSTEM = 0, MODE };
  char text[32] = "''"; //For quoted symbol

  int iSystem  = 0;
  char *pChars = 0L; // 0L will retrieve mode
  const char *pMode  = "error";
```

```
  if( int_from_num( &iSystem, args[ SYSTEM ] )) {
    if( nArgs == 2 ) {
      if( !char_ptr_from_string_or_symbol(
          &pChars, args[ MODE ] ) ) {
          pChars = 0L; // paranoia
    } }
    pMode = DoSorapsMode( iSystem, pChars );
    // returns a string from legacy C
  }

  strcpy( &text[2], pMode );
  return scheme_eval_string( text, sEnv );
  // e.g. "''stop" evals to a quoted symbol
}
```

The first object in `args` is an index to a telescope and the second is the desired control mode: `stop`, `vect`, etc. If the second argument is not provided, no attempt is made to change the mode. The primitive returns the final mode, in either case, as a quoted symbol suitable in expressions evaluated by the client. Internally, `DoSorapsMode` returns a string that represents the mode contained in the original C code. Typical primitives are more complex.

This simple primitive allows the control mode to be changed by any embedded Scheme program or any remote program that sends a `do-mode` expression. (A few more lines of code are needed to add `do-mode` to Scheme and bind it to the primitive.)

Serving Calculations A telescope system may include several subsystems that need information calculated by SORAPS. Rotators sometimes maintain orientation in a telescope's naturally rotating optical path, so they typically send an expression that uses the primitive `path-deg`; it contains arguments that specify the telescope and the location in the optical path where the rotation is needed. Other primitives perform time conversions or supply values such as range to satellites. These primitives are a work in progress. As requirements are added, primitives are written so that the capabilities are available to any s-expression that any client sends; this is more effective than writing specific messages for specific clients.

5.1.2 Bridge to Proprietary Network Libraries

Some systems are accessible only through nonstandard network libraries, so we embedded MzScheme in C programs that access those libraries. Primitives were then written to complete the bridge. This gives the other Scheme applications on the network a consistent way to access the bridged systems. These systems include a telescope focus controller, a dome controller, electro-pneumatic mirror actuators, and temperature sensors.

5.1.3 Hardware Access

We embedded MzScheme in several C programs that need to read and write electronic hardware in servos. Hardware includes analog output voltages, analog input voltages, parallel input ports, and bidirectional serial ports. Simple C programs were first written and debugged, then Scheme was embedded and furnished with primitive access to the hardware functions. The main software was then written in Scheme.

5.2 Motion Control Systems

We wrote servo software for a three-axis secondary mirror in Scheme. The program runs in MzScheme embedded in a small C program that accesses the hardware. A few primitives provide access to the input voltages, which indicate position, and the output voltages that drive the axes. Scheme reads the position voltages of the axes, calculates command voltages based on the position and the desired state, then sets the output voltages.

The program may access SORAPS to determine the range to the object and the elevation angle of the telescope. These values are used to adjust the focus and to maintain alignment between the primary and secondary mirrors. The servos also receive commands from user interfaces that are connected to the network.

The development process was remarkably efficient. On a suitable workstation, the servo program was developed remotely from DrScheme by using sockets to access the hardware primitives (i.e., the input and output voltages) on the servo computer (which is inconvenient for development.) When completed, the program was transferred to the servo computer and run in the embedded Scheme.

We also developed two optical-mechanical rotators, both prototyped in DrScheme. One interfaces custom hardware, while the other uses a serial port to interface a commercial motor driver.

5.3 S-expression Interface to Gimbals Servos

The gimbals for each of the three largest telescopes are controlled by a Networked Motion Control System, a C-program we developed for single board linux computers. An S-expression interface was developed for configuration, command, and networked control. NMCS periodically reads the encoders, computes the desired state vectors, computes the desired torques, drives the two axes, then processes s-expressions if any arrived over ethernet. These systems are markedly different than typical servos which are implemented with real time operating systems, digital signal processors, and rigid command and status interfaces. In NMCS, we use excess processor speed to run elegant interface software that can be accessed with anything that can read and write text over ethernet.

5.3.1 How S-Expressions are Used

The s-expression interface is used to configure the program, accept commands from SORAPS, and return servo status to SORAPS. Typical configuration values are position limits, rate limits, friction coefficients, and torque constants. Commands are required to periodically set the time, report the position of the sun, and provide state-vectors for the gimbals. A few examples demonstrate the flexibility of using s-expressions to specify one or more axes of the gimbals and to specify trajectories of variable order.

Commands Trajectories are specified as lists, and lists may contain lists for each axis:

```
(do-axes 0 '(t0 position velocity acceleration))
```

where 0 indicates axis-0, and the quantities represent numbers that describe a trajectory with a reference time of `t0`. Multiple axes are specified with a list of lists:

```
(do-axes '((t0 p0 v0 a0)(t1 p1 v1 a1)))
```

where all elements (`t0` etc.) represent numbers. For higher clarity, it is not necessary to list zeros in trajectories of lower order; the program integrates the available terms to calculate the position propagated from `t0`. For example, the following two expressions evaluate to the same fixed position for axis-1; the rate and acceleration are assumed to be zero in the second case:

```
(do-axes 1 '(12345678.01 45.0 0 0))
(do-axes 1 '(12345678.01 45.0))
```

The reference time, `t0`, is not needed for calculating fixed positions, but it is used to validate the command by testing if the trajectory time is within one second of the servo's time.

Status Lists of variable length are used to return status values that are ring-buffered each time the servo is serviced (typically every 20 milliseconds.) It is assumed that only one client is operating any telescope, so the primitives only return data that was buffered since

the previous request. This method allows the client program to casually query the servo for groups of information rather than forcing it to accept data as it is produced. An upper limit is established for the size of the replies in case the status was not recently requested.

For example, `(get-diffs)` returns a list of two lists. Each list contains the differences between the calculated and sensed positions for an axis. SORAPS queries about every quarter second and receives lists of about 12 values for each axis of each diagnostic value that was requested. These are used to compute statistics, display diagnostics on user interfaces, and to optimize the servo.

5.3.2 How NMCS Processes S-Expressions

The Small Fast S-Expression Library (SFSExp) is used to parse the s-expressions that NMCS receives over ethernet or reads from its configuration file. SFSExp was developed for a high speed cluster monitor at Los Alamos National Laboratories, so it easily handles the relatively low speed requirements for the motion control systems at the SOR.

We originally embedded MzScheme in NMCS, but garbage collections consumed around 10ms every 10 seconds or so; that was too marginal for the 20ms time slices needed by the servo. Rather than pursue a solution involving high priority interrupts or a real time operating system, the SFSExp library was employed to parse incoming expressions that essentially select C functions and call them with the remaining parameters.

Form for Data Modification and/or Access The only form implemented or needed in NMCS is

```
(list ['callback] (function1 parameter1...) ...)
```

where the optional callback function is intended to be defined on the client and the parameters of the functions cannot contain functions. All functions return s-expressions.

Functions return a value or a list; if parameters are supplied then the function attempts to set the values before returning the result. The gimbals have multiple axes, so lists-of-lists are converted into C arrays for each axis.

Destination of Replies NMCS supports a single socket that evaluates the incoming expressions and returns the result. The above message returns an expression that the client may evaluate:

```
(callback result1 ...)
```

The clients nearly always have a single receive thread that evaluates these responses. In other words, NMCS allows the client to call its own function with the results of NMCS functions. This behavior is compatible with the communications paradigm, described elsewhere in this document, that is more thoroughly implemented in MzScheme.

Tools for the Form We developed an API in C to provide a consistent way to set upper and lower limits on values and values in arrays. It also returns errors for illegal values or bad indices, for example. These features proved to be invaluable during the development of the servos because nearly all of the expressions involved passing numbers. Even these tiny subsets of language behavior are more useful than methods typically found in engineering: passing cryptic bits with rigid procedure calls, many without descriptive error messages.

5.4 Paradigm for Remote Command and Status

All communications between telescopes and their subsystems are conducted with s-expressions that can be evaluated with MzScheme or the s-expression interface of NMCS. The outgoing expressions produce incoming expressions that, when evaluated, cause a local function to be called with parameters that consist of the results of functions that were called on the remote system. This style allows concurrent access of multiple remote systems without requiring input decision trees or the management of multiple connections. When a programmer is working from a REPL, the callback mechanism is occasionally not used; in those cases the requesting function sends for a list of results by using a local function that blocks until it receives the result.

Communication takes place over UDP sockets whenever possible. Blocked TCP connections, whether due to network problems or software, are difficult for operators to solve because they often have no programming experience and little technical experience. A single unresolved timeout often leads to wholesale rebooting of computer systems if the connection can not be reestablished. Beginning with the implementation of PLT Scheme, nearly all connections are UDP, and all programs are written to be tolerant of incorrect or missing messages. We originally intended to write error detection functions for the adopted paradigm, but all of our activity takes place on a quality private network that is either working perfectly or not at all. This author observes that the connection problems caused by TCP far outweigh the packet error problems that they solve.

Typical programs use two sockets that may communicate with all of the remote systems. One socket sends all requests and evaluates any replies. The other socket evaluates any incoming requests then sends the replies. A catch-all exception handler was implemented after debugging was completed.

The functions shown below were used with version 209; slight changes are required for later releases of PLT Scheme, mainly due to unicode implementation.

Form for Requests Requests merely ask for a list of results of functions called on the remote application. The message form is described in Form for Data Modification and/or Access (for NMCS), but is much more generally useful in Scheme environments because no restrictions are placed on the expressions.

Send-Receive-Evaluate Requests can be sent with `udp-send` or `udp-send-to`, then incoming replies on the same socket are discovered and evaluated with a function that is either polled or looped in a blocking thread. The polled version is used in SORAPS because it is designed to wake up, check the socket, perform tasks, then sleep. On the other hand, scripts might use a blocking version if an operator is using commands in a REPL. The following function is intended to be polled, while a blocking version can be made by eliminating '*' from `udp-receive!*`.

```
(define make-function-to-eval ; accept udp socket
  (lambda( socket buffer-size )
    (define buffer (make-string buffer-size))
    (lambda()
      ; messages must fit in a single packet
      ; perhaps udp? should verify socket type
      (if (udp-bound? socket)
          (let ((n-rxed (call-with-values
                         (lambda()
                           (udp-receive!* socket
                                          buffer))
                         (lambda(n ip port) n))))
            (if n-rxed
                (with-handlers((exn? exn-message))
                  (eval
                   (read
                    (open-input-string
                     (substring
                      buffer 0 n-rxed)))))
                #f)) ; false instead of void
          #f)))) ; ditto
```

Receive-Evaluate-Reply The following form evaluates any request, then sends the reply. A simpler function uses MzScheme's `format` instead of the output string, but this function was developed first and has been used for several years.

```
(define make-function-to-eval-then-reply
  (lambda (socket buffer-size)
    (define buffer (make-string buffer-size))
    (lambda()
      (if (udp-bound? socket)
          (let-values
              (((n-rxed ip port)
                (udp-receive!* socket
                               buffer)))
            (if n-rxed
                (udp-send-to
                 socket ip port
                 (let ((o (open-output-string)))
                   (write
                    (with-handlers
                        ((exn? exn-message))
                      (eval
                       (read
                        (open-input-string
                         (substring
                          buffer 0 n-rxed))))) o)
                   (get-output-string o)))
                #f))
          #f))))
```

Multiple messages may be sent over the same socket because the replies may arrive in any order.

5.5 Scripting Environment for Experiments

DrScheme served as a scripting environment during development and operations of an experiment that required numerous motion control systems. This is described in Using Scheme to Script a Telescope System.

During operations required for the experiment, we were able to modify programs that were in use. For example, a telemetry thread in the DrScheme environment requests data via ethernet socket, processes the positions in the reply, then sends the results to SORAPS. When marginal reception was encountered, we developed and debugged a filter in a separate workspace. Genuine packets were taken from the running workspace to test the filter. When the filter was complete, we installed it and restarted the thread in a matter of seconds. Telescope operations were not interrupted.

We also over-wrote a measurement function while it was in use. Measurements from video tracking subsystems are scaled and rotated in Scheme before they are applied to SORAPS. The operation is complicated by the dynamic rotator that affects the apparent orientation of the tracker. A new tracker and rotator had unknown orientation, so we debugged the measurement function simply by editing and loading a file that overwrote the previous version. In the past, we rebuilt and restarted C code in the tracker for every iteration. Using Scheme, we were able to accomplish in one hour what previously required many.

5.6 Remote Development Environment

We used DrScheme for remote control and diagnostics while developing NMCS. Normal debugging techniques could not be used because such programs can not be arbitrarily suspended; the gimbals would "run away" if a breakpoint were encountered while torque was being applied. The environment consisted of threads which sent simulations of essential data that is normally sent from SORAPS. In the meantime, servo properties were changed by sending expressions from the REPL. The "viewport graphics" in MzScheme were used to display servo parameters while test functions moved the telescope along trajectories designed to be sensitive to parameters being adjusted.

We could have used one of several commercial development environments that we maintain, but they restrict both the hardware selection and the software techniques. On the other hand, NMCS is designed to be portable to anything that can support sockets and run programs built from C. The commercial environments are intended to run very fast, so they sacrifice software flexibility. Large telescopes cannot benefit from such speeds, so we do not believe performance could be gained by accepting the aforementioned restrictions. Furthermore, writing specific tools in DrScheme is arguably as fast as learning and using the commercial tools.

Threads and REPL Tests To prevent unsupervised telescope motion and prevent expensive damage due to slewing through the sun, three essential messages are periodically delivered to NMCS: the time, the desired trajectories of the axes, and the position of the sun. NMCS stops the gimbals if any of the messages are missed for more than a specified period; it uses the information to predict where the gimbals are going as well as the current location. Human action (on the SORAPS GUI) is then required to restart the gimbals to avoid an accident after a network dropout is resolved, for example. DrScheme was used to test these behaviors as KDevelop was used to debug NMCS.

To simulate SORAPS, three threads were started. One sent the position of the sun every 20 seconds, another sent the time every 10 seconds, and one sent gimbals vectors every second. From the REPL, the threads were individually suspended (while the gimbals were running) to ensure that NMCS stopped the gimbals. They were then restarted to ensure that NMCS did not start the gimbals without receiving other required commands that were also tested from the REPL.

Adjustment of Servo Properties We adjusted properties like torque constants, friction coefficients, and coefficients for parameter estimation while the telescope was following trajectories commanded by a thread that sent gimbals-vectors. Performance was optimized by viewing diagnostics displayed in the viewport window. These diagnostics included position, velocity, servo error, command difference, integrated torque, and timing errors.

Instead of creating large sets of functions and possibly GUI's to access them, we interactively wrote functions to change subsets of arguments during the servo optimization activities. For example, three acceleration constants are required by NMCS, but sometimes only one of them is adjusted:

```
(define (aa1 torque-per-bit) ;; aa1: adjust axis-1
 (send-to-nmcs
  (format
   "(list 'show (do-acceleration 1 '(1 2046 ~s)))"
   torque-per-bit)))
```

A receive thread evaluates the reply, causing the local function `show` to be called with the results of `do-acceleration` when it was called on NMCS. We intend to automate many of these optimization procedures, so this REPL approach forms a more appropriate foundation than a GUI.

The above expressions, especially ones that have variables, are sometimes assembled from lists rather than using `format`. It is arguably more sensible to do so, but some sort of formatting must eventually occur before the message is sent. We tend to form the strings as shown because they are blatantly readable.

Test Functions Test functions included sine-waves, constant rate dithering, and "racetracks". Trajectories were closed so that they

could be run indefinitely. This functional method contrasts typical servo development where files are "followed" after the gimbals are positioned before each diagnostic run. This technique was motivated by the use of functional programming: Commands are created from nested calls of functions whose only root variable is represented by a function, MzScheme's `current-milliseconds`.

6. Programming Techniques

This section describes many of the reusable techniques that were developed while working with MzScheme version 209.

6.1 Senders and Displayer

We wrote a simple displayer to show, in sequence, incoming and outgoing messages that access remote subsystems. `Make-sender` labels the diagnostics (to indicate their origin) and sends them to a program running in a separate DrScheme workspace (i.e., another window with a REPL.) Using a separate workspace prevents cluttering the operations REPL. A socket is implemented so that the displayer may also be hosted on a separate computer.

The displayer is normally a file that evaluates the following:

```
(letrec
    ((uos (udp-open-socket))
     (buffer (make-string 256))
     (lupe (lambda()
             (let-values
                 (((n ipa port)
                   (udp-receive! uos buffer)))
               (display (substring buffer 0 n))
               (newline))
             (lupe))))
  (udp-bind! uos #f 9999)
  (lupe))
```

The sender to each remote system is created with the function:

```
(define (make-sender soc ipa port prefix)
  (if (equal? prefix "")
      (lambda (text)
       (udp-send-to soc ipa port text))
      (lambda (text)
       (udp-send-to soc ipa port text)
       (udp-send-to
        soc "127.0.0.1" 9999
        (format "~a ~a" prefix text)))))
```

When the sender is created, non-empty text in `prefix` will cause all expressions passing through that sender to be displayed with the contents of `prefix`. The scripting environment also sends received expressions to the displayer, so that a clear ordering of messages is indicated in the window.

6.2 Simulating Transactions

We wrote a general transaction simulator before implementing ethernet communications. This simulator was used often:

```
(define (sim-transaction sleep-sec text)
  (thread
   (lambda()
     (sleep sleep-sec) ; simulate round-trip
     (eval
      (eval (read (open-input-string text)))))))
```

It first sleeps to simulate the expected round-trip delay, then evaluates the outgoing expression, and finally evaluates the result which is returned from a local simulation of the remote function. For example, the following will simulate stopping the gimbals:

```
(sim-transaction 1.5 "(list 'do-gimbals-mode
                            (do-mode corvus
                                     'stop))")
```

The simulation requires a local definition (a simulation) of the function `do-mode` and the definition of the telescope designator `corvus`. The callback function `do-gimbals-mode` is at the core of the local software that is being tested. Simulating the remote definitions also guided the creation of a clear, executable specification for the interface. For long delays, such as waiting several minutes for a telescope to move to a new location, the reference returned from `sim-transaction` was available for manipulating the thread.

6.3 Exception Handlers

We added exception handlers to the evaluator functions when we started using the new software. During development, it was better to let Scheme handle the exception and generate an error message. The exception handler is mainly used to prevent programs from halting due to misspelled or improperly formed expressions that are generated by new clients.

6.4 Connection Protocol

The communication paradigm relies on application level error control to compensate for the lack of detection and recovery provided by TCP-like protocols. To prevent the applications from halting, the message evaluators are wrapped in exception handlers which return any error message to the client. The motion control systems check messages by content; e.g., a rate of a thousand degrees per second is not accepted even if delivered without error. Most systems are designed to tolerate missed messages; e.g., a second order trajectory vector can be missed without noticeable errors in tracking. The clients nearly always use the response from a server to verify correct delivery; e.g., if a client sends a message to start a computational process in SORAPS, the response is used for verification.

We started to develop general techniques for error detection, such as echoing requests along with the responses, but we stopped for the lack of ways to cause or at least experience network errors.

6.5 Casual Polling of Remote Systems

Because TCP connections are avoided for practical reasons, we use an efficient technique for for getting uninterrupted repetitive data like telemetry. About every 8 seconds, the remote telemetry system is sent a message that requests data for ten seconds. This keeps the data flowing, it doesn't require the client to terminate the messages, yet it does not require a transaction for every message. A data message is not repeated after an overlapping request. Generally, the requests have the form:

```
(get-data 'callback-function seconds-to-send)
```

The server is required to send expressions of the form:

```
(callback-function the-data)
```

A related form is:

```
(get-data-if-available 'callback-function
                       within-the-next-n-seconds)
```

The server is required to send new data only if it is available within the next `n` seconds. This can be used when a telescope system is scanning for an object and it needs a camera to report when it firsts detects the object. The time limit prevents the camera from unexpectedly sending information at a much later time.

6.6 Message Timing

The arrival time of some expressions are stored in global variables. Typical functions that use these variables determine lateness and

provide re-triggering of events. For example, a task may keep the latest arrival time of a message, then compare it later to determine if a new one has arrived. Boolean techniques are not much simpler, and they do not contain enough information to determine elapsed time.

6.7 Making Specifications

A Scheme program needed access to a remote system written in C++, so we agreed that it must communicate via s-expressions. We wrote a Scheme simulation of the remote system and tested it with the Scheme client, then gave the simulation to the C++ programmer; no other specification was required because it was a working program written in an elegant language. We integrated and tested the complete system in about an hour, yet most of the hour was needed for the C++ programmer to debug socket software that he would not have needed to write had he used Scheme.

7. General Experience

How Scheme was Adopted I (the first author) was introduced to Scheme running on an Astronomer's Palm Pilot. Incapable of seeing usefulness, I dismissed Scheme until an email from the same person contained Scheme in one Day (SIOD), a pioneering Scheme environment. I obliged his suggestion to embed it in SORAPS at the same time that we were preparing to run the largest telescope remotely as it interacted with another telescope. Concurrently, another colleague showed me a paper on the Small Fast S-Expression Library because we were in search of a consistent way for our subsystems to communicate. By then, the potential of Scheme was obvious. I embedded MzScheme in SORAPS along with SIOD and gradually converted the few functions accessed by SIOD to MzScheme primitives. SIOD was finally removed, and I began writing a full suite of MzScheme primitives.

Reliability MzScheme v209 has been running continuously on 4 subsystems which are mostly single board computers running a diskless linux. One of them has been running for over a year, the rest are newer. Power failures and lightning strikes make it difficult to determine a "mean time before failure" that can be assigned to the subsystems; they normally are interrupted by such external events before they have a chance to fail.

The Small Fast S-expression Library has proven to be perfectly reliable as three have been running continuously for several years, only to be interrupted by power failures and lightning strikes. Ironically, the only maintenance needed in the first few years was a single-line fix of a memory leak caused by the first author... a leak that couldn't have occurred in a Scheme environment.

Language Issues Scheme statements have proven to be an effective basis for operating the systems. We have written thirty two primitives for SORAPS and a few dozen for the gimbals servos. Other subsystems have about a dozen. Error messages are a significant benefit of using Scheme. When a programmer incorrectly requests a message (especially from a REPL), an error message is returned which often reveals the problem. Typically, a parameter is forgotten, a Scheme error message is returned to the REPL, and the user solves the problem by reforming the message. When the serving program is implemented with MzScheme, the error messages are produced by the environment; the programmer is not required to develop them.

Elementary scripts have been written and used extensively for one experiment, they are planned for two more. Four scripts are needed for the current experiment, about four more will be required when all of the subsystems are complete. The scripting strategy has been unquestionably successful, but we will not improve it until we thoroughly study the possibilities... which include abandoning the approach and requiring users to learn basic Scheme.

We used closures as an effective way to realize object like behavior. The language requirements for any of the efforts in this paper are not overly complex, so adopting a much more complicated object oriented programming environment is probably not a good trade for the elegance of Scheme. The object system provided by PLT Scheme was not used in any of this work, mainly because of a lack of time to learn any of it.

Simulations We wrote simulations of nearly all subsystems; DrScheme was used to create executables for them. The simulations are used for developing scripts and to verify system operation before an experiment session begins. The balloon experiment has never been delayed by telescope systems because operations are tested with appropriate simulators before the experiment begins; electrical and mechanical problems are sometimes discovered and fixed. For these reasons, the combination of scripting and simulations are planned for all future experiments.

Foreign Function Interface vs Embedding Foreign Function Interfaces were not used in any of this work, mostly because the largest efforts were concentrated on SORAPS. Its C functions are so tightly tied to an event loop and graphical user interface that they are not appropriate for a Scheme interface. SORAPS functions are are being rewritten as primitives are added, so FFI's will be eventually be a viable option.

8. Conclusions

Scheme has significantly improved the efficiency of the two programmers who maintain telescope systems and prepare them for experiments. By using Scheme for configuration, communication, control, and at least elementary scripting, we are able to maintain the software for all systems and experiments in almost as little time as we previously needed for each. We modify SORAPS much less and the majority of configuration files have been eliminated because most configuration and all communication are contained in a few files of Scheme programs. We were able to create a single volume with directories containing software for all experiments and telescope systems; small Scheme programs load Scheme residing in the directories needed for given experiments.

Scripting the balloon experiment was valuable for two reasons. We successfully used the system at least once per month, yet another reason is perhaps more significant: Making scripts leads to better programs because the development process demands an accurate description of the problem. We were forced to answer two questions: What will be the sequence of events? What functions are needed to support them? The final few scripts for our experiment were so simple that they hardly seem worthy of discussion, but many iterations were required because problem definition is more difficult than many of us want to admit. This scripting effort directly contrasts our previous programs which rely entirely on graphical user interfaces. In those cases, we asked different questions: What GUI features are needed for fundamental operations? How can they be arranged so that operators can figure out how to conduct different experiments? Developing programs using the second approach is easier, but those programs depend on human expertise.

The s-expression communication paradigm allowed programmers to avoid using Scheme rather than encourage them to use it. SORAPS services messages sent from tracking systems, video displays, and timing devices which are written in C++, Java, and proprietary embedding languages. The lack of embedded Scheme in these systems significantly reduced development efficiency because each required the debugging of new communication software and new parsers; neither effort would have been required had Scheme been used. Most of our programmers (about 6) did not mind these difficulties, so they did not choose to adopt Scheme. Perhaps this is due to the fact that the basics of Scheme are easy to learn, then

programmers reject Scheme without realizing how much more it can offer.

We can gain much more utility from Scheme, even though the basics have contributed so positively. However, deciding where to spend development time is becoming more difficult. Significant wisdom is needed to understand the relationships between Scheme staples like modules, units, macros, and languages. Such knowledge is essential to the development of formal laboratory tools that could safely be used by people with diverse capabilities. An experienced computer scientist could contribute significantly to these efforts, but personnel in laboratories like ours need to be convinced that computer science can provide more than just programmers and compilers.

9. Future Effort

Future effort will include developing formal language layers for the controllers and experiments. Common functions have already been adopted, so a few layers of modules should guarantee common behavior.

Automatic optimization of servos and automatic calibration of gimbals pointing are also planned. While tracking stars and satellites, a Scheme program could observe control loop behavior and pointing corrections. From these observations, it could then update servo parameters and pointing models. These tasks are currently done manually.

References

[1] R. Q. Fugate, Air Force Phillips Lab; et. al. Two Generations of Laser Guidestar Adaptive Optics at the Starfire Optical Range. Journal of the Optical Society of America A II, 310-314 1994

[2] Matt Sottile, sexpr.sourceforge.net

[3] Matthew Flatt, Inside PLT Scheme 206.1

[4] Felleisen, Findler, Flatt, Krishnamurthi, How to Design Programs, MIT Press, Section 12.1

Gradual Typing for Functional Languages

Jeremy G. Siek
University of Colorado
siek@cs.colorado.edu

Walid Taha
Rice University
taha@rice.edu

Abstract

Static and dynamic type systems have well-known strengths and weaknesses, and each is better suited for different programming tasks. There have been many efforts to integrate static and dynamic typing and thereby combine the benefits of both typing disciplines in the same language. The flexibility of static typing can be improved by adding a type Dynamic and a typecase form. The safety and performance of dynamic typing can be improved by adding optional type annotations or by performing type inference (as in soft typing). However, there has been little formal work on type systems that allow a programmer-controlled migration between dynamic and static typing. Thatte proposed Quasi-Static Typing, but it does not statically catch all type errors in completely annotated programs. Anderson and Drossopoulou defined a nominal type system for an object-oriented language with optional type annotations. However, developing a sound, gradual type system for functional languages with structural types is an open problem.

In this paper we present a solution based on the intuition that the structure of a type may be partially known/unknown at compile-time and the job of the type system is to catch incompatibilities between the known parts of types. We define the static and dynamic semantics of a λ-calculus with optional type annotations and we prove that its type system is sound with respect to the simply-typed λ-calculus for fully-annotated terms. We prove that this calculus is type safe and that the cost of dynamism is "pay-as-you-go".

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs— *Type structure*

General Terms Languages, Performance, Theory

Keywords static and dynamic typing, optional type annotations

1. Introduction

Static and dynamic typing have different strengths, making them better suited for different tasks. Static typing provides early error detection, more efficient program execution, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.

The focus of this paper is languages that literally provide static and dynamic typing in the same program, with the programmer controlling the degree of static checking by annotating function parameters with types, or not. We use the term ***gradual typing*** for type systems that provide this capability. Languages that support gradual typing to a large degree include Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], and extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. The purpose of this paper is to provide a type-theoretic foundation for languages such as these with gradual typing.

There are numerous other ways to combine static and dynamic typing that fall outside the scope of gradual typing. Many dynamically typed languages have optional type annotations that are used to improve run-time performance but not to increase the amount of static checking. Common LISP [23] and Dylan [12, 37] are examples of such languages. Similarly, the Soft Typing of Cartwright and Fagan [7] improves the performance of dynamically typed languages but it does not statically catch type errors. At the other end of the spectrum, statically typed languages can be made more flexible by adding a Dynamic type and typecase form, as in the work by Abadi et al. [1]. However, such languages do not allow for programming in a dynamically typed style because the programmer is required to insert coercions to and from type Dynamic.

A short example serves to demonstrate the idea of gradual typing. Figure 1 shows a call-by-value interpreter for an applied λ-calculus written in Scheme extended with gradual typing and algebraic data types. The version on the left does not have type annotations, and so the type system performs little type checking and instead many tag-tests occur at run time.

As development progresses, the programmer adds type annotations to the parameters of interp, as shown on the right side of Figure 1, and the type system provides more aid in detecting errors. We use the notation ? for the dynamic type. The type system checks that the uses of env and e are appropriate: the case analysis on e is fine and so is the application of assq to x and env. The recursive calls to interp also type check and the call to **apply** type checks trivially because the parameters of **apply** are dynamic. Note that we are still using dynamic typing for the value domain of the object language. To obtain a program with complete static checking, we would introduce a datatype for the value domain and use that as the return type of interp.

Contributions We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated. These benefits include both safety and performance: type errors are caught at compile-time and values may be stored in unboxed form. That is, for statically typed portions of the program there is no need for run-time tags and tag checking.

We introduce a calculus named $\lambda^{?}_{\rightarrow}$ and define its type system (Section 2). We show that this type system, when applied to fully an-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

```
(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

```
(type expr (datatype (Var ,symbol)
                     (Int ,int)
                     (App ,expr ,expr)
                     (Lam ,symbol ,expr)
                     (Succ ,expr)))
(type envty (listof (pair symbol ?)))

(define interp
  (λ ((env : envty) (e : expr))
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

Figure 1. An example of gradual typing: an interpreter with varying amounts of type annotations.

notated terms, is equivalent to that of the simply-typed lambda calculus (Theorem 1). This property ensures that for fully-annotated programs all type errors are caught at compile-time. Our type system is the first gradual type system for structural types to have this property. To show that our approach to gradual typing is suitable for imperative languages, we extend $\lambda^{?}_{\rightarrow}$ with ML-style references and assignment (Section 4).

We define the run-time semantics of $\lambda^{?}_{\rightarrow}$ via a translation to a simply-typed calculus with explicit casts, $\lambda^{\langle\tau\rangle}_{\rightarrow}$, for which we define a call-by-value operational semantics (Section 5). When applied to fully-annotated terms, the translation does not insert casts (Lemma 4), so the semantics exactly matches that of the simply-typed λ-calculus. The translation preserves typing (Lemma 3) and $\lambda^{\langle\tau\rangle}_{\rightarrow}$ is type safe (Lemma 8), and therefore $\lambda^{?}_{\rightarrow}$ is type safe: if evaluation terminates, the result is either a value of the expected type or a cast error, but never a type error (Theorem 2).

On the way to proving type safety, we prove Lemma 5 (Canonical Forms), which is of particular interest because it shows that the run-time cost of dynamism in $\lambda^{?}_{\rightarrow}$ can "pay-as-you-go". Run-time polymorphism is restricted to values of type ?, so for example, a value of type int must actually be an integer, whereas a value of type ? may contain an integer or a Boolean or anything at all. Compilers for $\lambda^{?}_{\rightarrow}$ may use efficient, unboxed, representations for values of ground and function type, achieving the performance benefits of static typing for the parts of programs that are statically typed.

The proofs of the lemmas and theorems in this paper were written in the Isar proof language [28, 42] and verified by the Isabelle proof assistant [29]. We provide proof sketches in this paper and the full proofs are available in the companion technical report [39]. The statements of the definitions (including type systems and semantics), lemmas, propositions, and theorems in this paper were automatically generated from the Isabelle files. Free variables that appear in these statements are universally quantified.

2. Introduction to Gradual Typing

The gradually-typed λ-calculus, $\lambda^{?}_{\rightarrow}$, is the simply-typed λ-calculus extended with a type ? to represent dynamic types. We present gradual typing in the setting of the simply-typed λ-calculus to reduce unnecessary distractions. However, we intend to show how gradual typing interacts with other common language features, and as a first step combine gradual typing with ML-style references in Section 4.

Syntax of the Gradually-Typed Lambda Calculus $e \in \lambda^{?}_{\rightarrow}$

Variables	$x \in \mathbb{X}$		
Ground Types	$\gamma \in \mathbb{G}$		
Constants	$c \in \mathbb{C}$		
Types	τ	$::=$	$\gamma \mid ? \mid \tau \rightarrow \tau$
Expressions	e	$::=$	$c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e$
			$\lambda x.\ e \equiv \lambda x{:}?.\ e$

A procedure without a parameter type annotation is syntactic sugar for a procedure with parameter type ?.

The main idea of our approach is the notion of a type whose structure may be partially known and partially unknown. The unknown portions of a type are indicated by ?. So, for example, the type **number** $*$? is the type of a pair whose first element is of type **number** and whose second element has an unknown type. To program in a dynamically typed style, omit type annotations on parameters; they are by default assigned the type ?. To enlist more help from the type checker, add type annotations, possibly with ? occurring inside the types to retain some flexibility.

The job of the static type system is to reject programs that have inconsistencies in the known parts of types. For example, the program

```
((λ (x : number) (succ x)) #t) ;; reject
```

should be rejected because the type of #t is not consistent with the type of the parameter x, that is, **boolean** is not consistent with **number**. On the other hand, the program

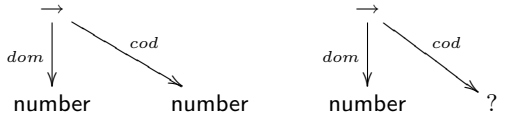
```
((λ (x) (succ x)) #t) ;; accept
```

should be accepted by the type system because the type of x is considered unknown (there is no type annotation) and therefore not within the realm of static checking. Instead, the type error will be caught at run-time (as is typical of dynamically typed languages), which we describe in Section 5.

As usual things become more interesting with first class procedures. Consider the following example of mapping a procedure over a list.

```
map : (number → number) * number list → number list
(map (λ (x) (succ x)) (list 1 2 3)) ;; accept
```

The **map** procedure is expecting a first argument whose type is **number** → **number** but the argument (λ(x) (succ x)) has type ? → **number**. We would like the type system to accept this program, so how should we define consistency for procedure types? The intuition is that we should require the known portions of the two types to be equal and ignore the unknown parts. There is a useful analogy with the mathematics of partial functions: two partial functions are ***consistent*** when every elements that is in the domain of both functions is mapped to the same result. This analogy can be made formal by considering types as trees [32].



Trees can be represented as partial functions from paths to node labels, where a path is a sequence of edge labels: $[l_1, \ldots, l_n]$. The above two trees are the following two partial functions f and g. We interpret unknown portions of a type simply as places where the partial function is undefined. So, for example, g is undefined for the path $[cod]$.

$$\begin{aligned} f([]) &= \rightarrow \\ f([dom]) &= number \\ f([cod]) &= number \end{aligned}$$

$$\begin{aligned} g([]) &= \rightarrow \\ g([dom]) &= number \end{aligned}$$

The partial functions f and g are consistent because they produce the same output for the inputs [] and $[dom]$.

We axiomatize the consistency relation $\sim$ on types with the following definition.

Type Consistency $\tau \sim \tau$

$$\text{(CRefl)}\ \tau \sim \tau \qquad \text{(CFun)}\ \frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{\sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2}$$

$$\text{(CUnR)}\ \tau \sim ? \qquad \text{(CUnL)}\ ? \sim \tau$$

The type consistency relation is reflexive and symmetric but not transitive (just like consistency of partial functions).

Proposition 1.

- $\tau \sim \tau$
- If $\sigma \sim \tau$ then $\tau \sim \sigma$.
- $\neg\,(\forall\, \tau_1\ \tau_2\ \tau_3.\ \tau_1 \sim \tau_2 \wedge \tau_2 \sim \tau_3 \longrightarrow \tau_1 \sim \tau_3)$

Our gradual type system is shown in Figure 2. The environment Γ is a function from variables to optional types ($\lfloor\tau\rfloor$ or $\perp$). The type system is parameterized on a signature Δ that assigns types to constants. The rules for variables, constants, and functions are standard. The first rule for function application (GApp1) handles the case when the function type is unknown. The argument may have any type and the resulting type of the application is unknown. The second rule for function application (GApp2) handles when the function type is known and allows an argument whose type is consistent with the function's parameter type.

Figure 2. A Gradual Type System

$\boxed{\Gamma \vdash_G e : \tau}$

$$\text{(GVar)}\ \frac{\Gamma\, x = \lfloor\tau\rfloor}{\Gamma \vdash_G x : \tau} \qquad \text{(GConst)}\ \frac{\Delta\, c = \tau}{\Gamma \vdash_G c : \tau}$$

$$\text{(GLam)}\ \frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda\, x{:}\sigma.\ e : \sigma \rightarrow \tau}$$

$$\text{(GApp1)}\ \frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\ e_2 : ?}$$

$$\text{(GApp2)}\ \frac{\Gamma \vdash_G e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash_G e_1\ e_2 : \tau'}$$

Relation to the untyped λ-calculus We would like our gradual type system to accept all terms of the untyped λ-calculus (all unannotated terms), but it is not possible to simultaneously achieve this and provide type safety for fully-annotated terms. For example, suppose there is a constant succ with type **number** → **number**. The term (succ "hi") has no type annotations but it is also fully annotated because there are no function parameters to annotate. The type system must either accept or reject this program. We choose to reject. Of course, if succ were given the type ? → ? then (succ "hi") would be accepted. In any event, our gradual type system provides the same expressiveness as the untyped λ-calculus. The following translation converts any λ-term into an observationally equivalent well-typed term of $\lambda^{?}_{\rightarrow}$.

$$\begin{aligned} [\![c]\!] &= c \\ [\![x]\!] &= x \\ [\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\ [\![e_1\ e_2]\!] &= ((\lambda x.x)[\![e_1]\!])[\![e_2]\!] \end{aligned}$$

Relation to the simply-typed λ-calculus Let $\lambda_{\rightarrow}$ denote the terms of the simply-typed λ-calculus and let $\Gamma \vdash_{\rightarrow} e : \tau$ stand for the standard typing judgment of the simply-typed λ-calculus. For terms in $\lambda_{\rightarrow}$ our gradual type system is equivalent to simple typing.

Theorem 1 (Equivalence to simple typing for fully-annotated terms)**.** If $e \in \lambda_{\rightarrow}$ then $\emptyset \vdash_G e : \tau = \emptyset \vdash_{\rightarrow} e : \tau$.

Proof Sketch. The rules for our gradual type system are the same as for the STLC if one removes the rules that mention ?. The type compatibility relation collapses to type equality once all rules involving ? are removed. □

A direct consequence of this equivalence is that our gradual type system catches the same static errors as the type system for $\lambda_{\rightarrow}$.

Corollary 1 (Full static error detection for fully-annotated terms)**.** If $e \in \lambda_{\rightarrow}$ and $\nexists \tau.\ \emptyset \vdash_{\rightarrow} e : \tau$ then $\nexists \tau'.\ \emptyset \vdash_G e : \tau'$. *(This is just the contrapositive of soundness.)*

Before describing the run-time semantics of $\lambda^?_\rightarrow$ we compare our type system for $\lambda^?_\rightarrow$ with an alternative design based on subtyping.

3. Comparison with Quasi-Static Typing

Our first attempt to define a gradual type system was based on Thatte's quasi-static types [40]. Thatte uses a standard subtyping relation $<:$ with a top type Ω to represent the dynamic type. As before, the meta-variable γ ranges over ground types such as **number** and **boolean**.

Subtyping rules. $\tau <: \tau'$

$$\frac{}{\gamma <: \gamma} \qquad \frac{}{\tau <: \Omega} \qquad \frac{\sigma_1 <: \tau_1 \quad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}$$

The quasi-static type system includes the usual subsumption rule.

$$\text{QSUB}\ \frac{\Gamma \vdash e : \tau \quad \tau <: \sigma}{\Gamma \vdash e : \sigma}$$

Subsumption allows programs such as the following to type check by allowing implicit up-casts. The value #t of type **boolean** is up-cast to Ω, the type of the parameter x.

```
((λ (x) ...) #t) ;; ok, boolean <: Ω
```

However, the subsumption rule will not allow the following program to type check. The addition operator expects type **number** but gets an argument of type Ω.

```
(λ (x) (succ x))
```

Thatte's solution for this is to also allow an implicit down-cast in the (QAPP2) rule for function application.

$$(\text{QAPP2})\ \frac{\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Gamma \vdash e_2 : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1\ e_2) : \sigma'}$$

Unfortunately, the subsumption rule combined with (QAPP2) allows too many programs to type check for our taste. For example, we can build a typing derivation for the following program, even though it was rejected by our gradual type system.

```
((λ (x : number) (succ x)) #t)
```

The subsumption rule allows #t to be implicitly cast to Ω and then the above rule for application implicitly casts Ω down to **number**.

To catch errors such as these, Thatte added a second phase to the type system called plausibility checking. This phase rewrites the program by collapsing sequences of up-casts and down-casts and signals an error if it encounters a pair of casts that together amount to a "stupid cast"[22], that is, casts that always fail because the target is incompatible with the subject.

Figure 3 shows Thatte's Quasi-Static type system. The judgment $\Gamma \vdash e \Rightarrow e' : \tau$ inserts up-casts and down-casts and the judgment $e \rightsquigarrow e'$ collapses sequences of casts and performs plausibility checking. The type system is parameterized on the function Δ mapping constants to types. The environment Γ is a function from variables to optional types ($\lfloor\tau\rfloor$ or $\bot$).

Subsumption rules are slippery, and even with the plausibility checks the type system fails to catch many errors. For example, there is still a derivation for the program

```
((λ (x : number) (succ x)) #t)
```

The reason is that both the operator and operand may be implicitly up-cast to Ω. The rule (QAPP1) then down-casts the operator to $\Omega \to \Omega$. Plausibility checking succeeds because there is a greatest lower bound of **number** $\to$ **number** and $\Omega \to \Omega$, which is $\Omega \to$ **number**. So the quasi-static system fails to statically catch the type error.

Figure 3. Thatte's Quasi-Static Typing.

$\boxed{\Gamma \vdash e \Rightarrow e' : \tau}$

$$(\text{QVAR})\ \frac{\Gamma\ x = \lfloor\tau\rfloor}{\Gamma \vdash x \Rightarrow x : \tau}$$

$$(\text{QCONST})\ \frac{\Delta\ c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$$

$$(\text{QLAM})\ \frac{\Gamma, x : \tau \vdash e \Rightarrow e' : \sigma}{\Gamma \vdash (\lambda x{:}\tau.\ e) \Rightarrow (\lambda x{:}\tau.\ e') : \tau \to \sigma}$$

$$(\text{QSUB})\ \frac{\Gamma \vdash e \Rightarrow e' : \tau \quad \tau <: \sigma}{\Gamma \vdash e \Rightarrow e' \uparrow^\sigma_\tau : \sigma}$$

$$(\text{QAPP1})\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \Omega \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash (e_1\ e_2) \Rightarrow ((e_1' \downarrow^\Omega_{\tau\to\Omega})\ e_2') : \Omega}$$

$$(\text{QAPP2})\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \sigma \to \sigma' \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1\ e_2) \Rightarrow (e_1'\ (e_2' \downarrow^\tau_\sigma)) : \sigma'}$$

$\boxed{e \rightsquigarrow e'}$

$$e \downarrow^\tau_\tau \rightsquigarrow e \qquad e \uparrow^\tau_\tau \rightsquigarrow e$$

$$e \downarrow^\tau_\sigma \downarrow^\sigma_\mu \rightsquigarrow e \downarrow^\tau_\mu \qquad e \uparrow^\sigma_\mu \uparrow^\tau_\sigma \rightsquigarrow e \uparrow^\tau_\mu$$

$$\frac{\mu = \tau \sqcap \nu}{e \uparrow^\sigma_\tau \downarrow^\sigma_\nu \rightsquigarrow e \downarrow^\tau_\mu \uparrow^\nu_\mu} \qquad \frac{\nexists \mu.\mu = \tau \sqcap \nu}{e \uparrow^\sigma_\tau \downarrow^\sigma_\nu \rightsquigarrow \mathsf{wrong}}$$

As noted by Oliart [30], Thatte's quasi-static type system does not correspond to his type checking *algorithm* (Theorem 7 of [40] is incorrect). Thatte's type checking algorithm does not suffer from the above problems because the algorithm does not use the subsumption rule and instead performs all casting at the application rule, disallowing up-casts to Ω followed by arbitrary down-casts. Oliart defined a simple syntax-directed type system that is equivalent to Thatte's algorithm, but did not state or prove any of its properties. We initially set out to prove type safety for Oliart's subtype-based type system, but then realized that the consistency relation provides a much simpler characterization of when implicit casts should be allowed.

At first glance it may seem odd to use a symmetric relation such as consistency instead of an anti-symmetric relation such as subtyping. There is an anti-symmetric relation that is closely related to consistency, the usual partial ordering relation for partial functions: $f \sqsubseteq g$ if the graph of f is a subset of the graph of g. (Note that the direction is flipped from that of the subtyping relation $<:$, where greater means less information.) A cast from τ to σ, where $\sigma \sqsubseteq \tau$, always succeeds at run-time as we are just hiding type information by replacing parts of a type with ?. On the other hand, a cast from σ to τ may fail because the run-time type of the value may not be consistent with τ. The main difference between $\sqsubseteq$ and $<:$ is that $\sqsubseteq$ is covariant for the domain of a procedure type, whereas $<:$ is contra-variant for the domain of a procedure type.

Figure 4. Type Rules for References

$\boxed{\Gamma \vdash_G e : \tau}$

(GREF) $$\frac{\Gamma \vdash_G e : \tau}{\Gamma \vdash_G \mathit{ref}\ e : \mathit{ref}\ \tau}$$

(GDEREF1) $$\frac{\Gamma \vdash_G e : ?}{\Gamma \vdash_G\ !e : ?}$$

(GDEREF2) $$\frac{\Gamma \vdash_G e : \mathit{ref}\ \tau}{\Gamma \vdash_G\ !e : \tau}$$

(GASSIGN1) $$\frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \mathit{ref}\ \tau}$$

(GASSIGN2) $$\frac{\Gamma \vdash_G e_1 : \mathit{ref}\ \tau \qquad \Gamma \vdash_G e_2 : \sigma \qquad \sigma \sim \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \mathit{ref}\ \tau}$$

4. Gradual Typing and References

It is often challenging to integrate type system extensions with imperative features such as references with assignment. In this section we extend the calculus to include ML-style references. The following grammar shows the additions to the syntax.

Adding references to $\lambda^{?}_{\rightarrow}$

Types	τ	::=	$\ldots \mid \mathit{ref}\ \tau$
Expressions	e	::=	$\ldots \mid \mathit{ref}\ e \mid\ !e \mid e \leftarrow e$

The form $\mathit{ref}\ \ e$ creates a reference cell and initializes it with the value that results from evaluating expression e. The dereference form $!e$ evaluates e to the address of a location in memory (hopefully) and returns the value stored there. The assignment form $e \leftarrow e$ stores the value form the right-hand side expression in the location given by the left-hand side expression.

Figure 4 shows the gradual typing rules for these three new constructs. In the (GASSIGN2) we allow the type of the right-hand side to differ from the type in the left-hand's reference, but require the types to be compatible. This is similar to the (GAPP2) rule for function application.

We do not change the definition of the consistency relation, which means that references types are invariant with respect to consistency. The reflexive axiom $\tau \sim \tau$ implies that $\mathit{ref}\ \tau \sim \mathit{ref}\ \tau$. The situation is analogous to that of the combination of references with subtyping [32]: allowing variance under reference types compromises type safety. The following program demonstrates how a covariant rule for reference types would allow type errors to go uncaught by the type system.

```
let r1 = ref (λ y. y) in
let r2 : ref ? = r1 in
  r2 ← 1;
  !r1 2
```

The reference r1 is initialized with a function, and then r2 is aliased to r1, using the covariance to allow the change in type to ref ?. We can then write an integer into the cell pointed to by r2 (and by r1). The subsequent attempt to apply the contents of r1 as if it were a function fails at runtime.

5. Run-time semantics

We define the semantics for $\lambda^{?}_{\rightarrow}$ in two steps. We first define a cast insertion translation from $\lambda^{?}_{\rightarrow}$ to an intermediate language with explicit casts which we call $\lambda^{\langle\tau\rangle}_{\rightarrow}$. We then define a call-by-value operational semantics for $\lambda^{\langle\tau\rangle}_{\rightarrow}$. The explicit casts have the syntactic form $\langle\tau\rangle e$ where τ is the target type. When e evaluates to v, the cast will check that the type of v is consistent with τ and then produce a value based on v that has the type τ. If the type of v is inconsistent with τ, the cast produces a *CastError*. The intuition behind this kind of cast is that it reinterprets a value to have a different type either by adding or removing type information.

The syntax of $\lambda^{\langle\tau\rangle}_{\rightarrow}$ extends that of $\lambda^{?}_{\rightarrow}$ by adding a cast expression.

Syntax of the intermediate language. $e \in \lambda^{\langle\tau\rangle}_{\rightarrow}$

Expressions e ::= $\ldots \mid \langle\tau\rangle e$

5.1 Translation to $\lambda^{\langle\tau\rangle}_{\rightarrow}$.

The cast insertion judgment, defined in Figure 5, has the form $\Gamma \vdash e \Rightarrow e' : \tau$ and mimics the structure of our gradual typing judgment of Figure 2. It is trivial to show that these two judgments accept the same set of terms. We presented the gradual typing judgment separately to provide an uncluttered *specification* of well-typed terms. In Figure 5, the rules for variables, constants, and functions are straightforward. The first rule for application (CAPP1) handles the case when the function has type ? and inserts a cast to $\tau_2 \rightarrow ?$ where τ_2 is the argument's type. The second rule for application (CAPP2) handles the case when the function's type is known and the argument type differs from the parameter type, but is consistent. In this case the argument is cast to the parameter type τ. We could have instead cast the function; the choice was arbitrary. The third rule for application (CAPP3) handles the case when the function type is known and the argument's type is identical to the parameter type. No casts are needed in this case. The rules for reference assignment are similar to the rules for application. However, for CASSIGN2 the choice to cast the argument and not the reference is because we need references to be invariant to preserve type soundness.

Next we define a type system for the intermediate language $\lambda^{\langle\tau\rangle}_{\rightarrow}$. The typing judgment has the form $\Gamma|\Sigma \vdash e : \tau$. The Σ is a store typing: it assigns types to memory locations. The type system, defined in Figure 6, extends the STLC with a rule for explicit casts. The rule (TCAST) requires the expression e to have a type consistent with the target type τ.

The inversion lemmas for $\lambda^{\langle\tau\rangle}_{\rightarrow}$ are straightforward.

Lemma 1 (Inversion on typing rules.)**.**

1. If $\Gamma \mid \Sigma \vdash x : \tau$ then $\Gamma\, x = \lfloor\tau\rfloor$.
2. If $\Gamma \mid \Sigma \vdash c : \tau$ then $\Delta\, c = \tau$.
3. If $\Gamma \mid \Sigma \vdash \lambda\, x{:}\sigma.\, e : \tau$ then $\exists \tau'.\, \tau = \sigma \rightarrow \tau'$.
4. If $\Gamma \mid \Sigma \vdash e_1\ e_2 : \tau'$ then $\exists \tau.\, \Gamma \mid \Sigma \vdash e_1 : \tau \rightarrow \tau' \wedge \Gamma \mid \Sigma \vdash e_2 : \tau$.
5. If $\Gamma \mid \Sigma \vdash \langle\sigma\rangle\, e : \tau$ then $\exists \tau'.\, \Gamma \mid \Sigma \vdash e : \tau' \wedge \sigma = \tau \wedge \tau' \sim \sigma$.
6. If $\Gamma \mid \Sigma \vdash \mathit{ref}\, e : \mathit{ref}\ \tau$ then $\Gamma \mid \Sigma \vdash e : \tau$.
7. If $\Gamma \mid \Sigma \vdash\ !e : \tau$ then $\Gamma \mid \Sigma \vdash e : \mathit{ref}\ \tau$.
8. If $\Gamma \mid \Sigma \vdash e_1 \leftarrow e_2 : \mathit{ref}\ \tau$ then $\Gamma \mid \Sigma \vdash e_1 : \mathit{ref}\ \tau \wedge \Gamma \mid \Sigma \vdash e_2 : \tau$.

Figure 5. Cast Insertion

$$\boxed{\Gamma \vdash e \Rightarrow e' : \tau}$$

$$\text{(CVar)}\quad \frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \vdash x \Rightarrow x : \tau}$$

$$\text{(CConst)}\quad \frac{\Delta\, c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$$

$$\text{(CLam)}\quad \frac{\Gamma(x \mapsto \sigma) \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \lambda\, x{:}\sigma.\, e \Rightarrow \lambda\, x{:}\sigma.\, e' : \sigma \to \tau}$$

$$\text{(CApp1)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow (\langle \tau_2 \to ? \rangle\ e'_1)\ e'_2 : ?}$$

$$\text{(CApp2)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2 \qquad \tau_2 \neq \tau \qquad \tau_2 \sim \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e'_1\ (\langle \tau \rangle\ e'_2) : \tau'}$$

$$\text{(CApp3)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow e'_1\ e'_2 : \tau'}$$

$$\text{(CRef)}\quad \frac{\Gamma \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \mathit{ref}\ e \Rightarrow \mathit{ref}\ e' : \mathit{ref}\ \tau}$$

$$\text{(CDeref1)}\quad \frac{\Gamma \vdash e \Rightarrow e' : ?}{\Gamma \vdash\ !e \Rightarrow\ !(\langle \mathit{ref}\ ? \rangle\ e') : ?}$$

$$\text{(CDeref2)}\quad \frac{\Gamma \vdash e \Rightarrow e' : \mathit{ref}\ \tau}{\Gamma \vdash\ !e \Rightarrow\ !e' : \tau}$$

$$\text{(CAssign1)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow (\langle \mathit{ref}\ \tau_2 \rangle\ e'_1) \leftarrow e'_2 : \mathit{ref}\ \tau_2}$$

$$\text{(CAssign2)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \mathit{ref}\ \tau \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \sigma \qquad \sigma \neq \tau \qquad \sigma \sim \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow (\langle \tau \rangle\ e'_2) : \mathit{ref}\ \tau}$$

$$\text{(CAssign3)}\quad \frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \mathit{ref}\ \tau \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow e'_2 : \mathit{ref}\ \tau}$$

Proof Sketch. They are proved by case analysis on the type rules. □

The type system for $\lambda_{\rightarrow}^{\langle\tau\rangle}$ is deterministic: it assigns a unique type to an expression given a fixed environment.

Lemma 2 (Unique typing)**.** If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e : \tau'$ then $\tau = \tau'$.

Proof Sketch. The proof is by induction on the typing derivation and uses the inversion lemmas. □

The cast insertion translation, if successful, produces well-typed terms of $\lambda_{\rightarrow}^{\langle\tau\rangle}$.

Lemma 3. If $\Gamma \vdash e \Rightarrow e' : \tau$ then $\Gamma \mid \emptyset \vdash e' : \tau$.

Figure 6. Type system for the intermediate language $\lambda_{\rightarrow}^{\langle\tau\rangle}$

$$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$$

$$\text{(TVar)}\quad \frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \mid \Sigma \vdash x : \tau}$$

$$\text{(TConst)}\quad \frac{\Delta\, c = \tau}{\Gamma \mid \Sigma \vdash c : \tau}$$

$$\text{(TLam)}\quad \frac{\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \lambda\, x{:}\sigma.\, e : \sigma \to \tau}$$

$$\text{(TApp)}\quad \frac{\Gamma \mid \Sigma \vdash e_1 : \tau \to \tau' \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1\ e_2 : \tau'}$$

$$\text{(TCast)}\quad \frac{\Gamma \mid \Sigma \vdash e : \sigma \qquad \sigma \sim \tau}{\Gamma \mid \Sigma \vdash \langle \tau \rangle\ e : \tau}$$

$$\text{(TRef)}\quad \frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \mathit{ref}\ e : \mathit{ref}\ \tau}$$

$$\text{(TDeref)}\quad \frac{\Gamma \mid \Sigma \vdash e : \mathit{ref}\ \tau}{\Gamma \mid \Sigma \vdash\ !e : \tau}$$

$$\text{(TAssign)}\quad \frac{\Gamma \mid \Sigma \vdash e_1 : \mathit{ref}\ \tau \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 \leftarrow e_2 : \mathit{ref}\ \tau}$$

$$\text{(TLoc)}\quad \frac{\Sigma\, l = \lfloor \tau \rfloor}{\Gamma \mid \Sigma \vdash l : \mathit{ref}\ \tau}$$

Proof Sketch. The proof is by induction on the cast insertion derivation. □

When applied to terms of $\lambda_{\rightarrow}$, the translation is the identity function, i.e., no casts are inserted. [1]

Lemma 4. If $\emptyset \vdash e \Rightarrow e' : \tau$ and $e \in \lambda_{\rightarrow}$ then $e = e'$.

Proof Sketch. The proof is by induction on the cast insertion derivation. □

When applied to terms of the untyped λ-calculus, the translation inserts just those casts necessary to prevent type errors from occuring at run-time, such as applying a non-function.

5.2 Run-time semantics of $\lambda_{\rightarrow}^{\langle\tau\rangle}$.

The following grammar describes the results of evaluation: the result is either a value or an error, where values are either a simple value (variables, constants, functions, and locations) or a simple value enclosed in a single cast, which serves as a syntacical representation of boxed values.

[1] This lemma is for closed terms (this missing Γ means an empty environment). A similar lemma is true of open terms, but we do not need the lemma for open terms and the statement is more complicated because there are conditions on the environment.

Values, Errors, and Results

Locations	$l \in \mathbb{L}$		
Simple Values	$s \in \mathbb{S}$	$::=$	$x \mid c \mid \lambda x : \tau.e \mid l$
Values	$v \in \mathbb{V}$	$::=$	$s \mid \langle ? \rangle s$
Errors	ε	$::=$	*CastError* $\mid$ *TypeError* $\mid$ *KillError*
Results	r	$::=$	$v \mid \varepsilon$

It is useful to distinguish two different kinds of run-time type errors. In weakly typed languages, type errors result in undefined behavior, such as causing a segmentation fault or allowing a hacker to create a buffer overflow. We model this kind of type error with *TypeError*. In strongly-typed dynamic languages, there may still be type errors, but they are caught by the run-time system and do not cause undefined behavior. They typically cause the program to terminate or else raise an exception. We model this kind of type error with *CastError*. The *KillError* is a technicality pertaining to the type safety proof that allows us to prove a form of "progress" in the setting of a big-step semantics.

We define simple function values (*SimpleFunVal*) to contain lambda abstractions and functional constants (such as succ), and function values (*FunVal*) include simple function values and simple function values cast to ?.

As mentioned in Section 1, the Canonical Forms Lemma is of particular interest due to its implications for performance. When an expression has either ground or function type (not ?) the kind of resulting value is fixed, and a compiler may use an efficient unboxed representation. For example, if an expression has type int, then it will evaluate to a value of type int (by the forthcoming Soundness Lemma 8) and then the Canonical Forms Lemma tells us that the value must be an integer.

Lemma 5 (Canonical Forms)**.**

- If $\emptyset \mid \Sigma \vdash v : \mathit{int}$ and $v \in \mathbb{V}$ then $\exists n.\, v = n$.
- If $\emptyset \mid \Sigma \vdash v : \mathit{bool}$ and $v \in \mathbb{V}$ then $\exists b.\, v = b$.
- If $\emptyset \mid \Sigma \vdash v : ?$ and $v \in \mathbb{V}$ then $\exists v'.\, v = \langle ? \rangle\, v' \wedge v' \in \mathbb{S}$.
- If $\emptyset \mid \Sigma \vdash v : \tau \to \tau'$ and $v \in \mathbb{V}$ then $v \in \mathit{SimpleFunVal}$.
- If $\emptyset \mid \Sigma \vdash v : \mathit{ref}\ \tau$ and $v \in \mathbb{V}$ then $\exists l.\, v = l \wedge \Sigma\, l = \lfloor \tau \rfloor$.

Proof Sketch. They are proved using the inversion lemmas and case analysis on values. □

We define the run-time semantics for $\lambda^{\langle\tau\rangle}_{\rightarrow}$ in big-step style with substitution and not environments. Substitution, written $[x := e]e$, is formalized in the style of Curry [3], where bound variables are α-renamed during substitution to avoid the capture of free variables.

The evaluation judgment has the form $e \hookrightarrow_n r$, where e evaluates to the result r with a derivation depth of n. The derivation depth is used to force termination so that derivations can be constructed for otherwise non-terminating programs [11]. The n-depth evaluation allows Lemma 8 (Soundness) to distinguish between terminating and non-terminating programs. We will say more about this when we get to Lemma 8.

The evaluation rules, shown in Figures 7 and 8, are the standard call-by-value rules for the λ-calculus [33] with additional rules for casts and a special termination rule. We parameterize the semantics over the function δ which defines the behavior of functional constants and is used in rule (EDELTA). The helper function *unbox* removes an enclosing cast from a value, if there is one.

$$\begin{aligned} \mathit{unbox}\ s &= s \\ \mathit{unbox}\ (\langle\tau\rangle\ s) &= s \end{aligned}$$

The evaluation rules treat the cast expression like a boxed, or tagged, value. It is straightforward to define a lower-level semantics that explicitly tags every value with its type (the full type, not just the top level constructor) and then uses these type representations instead of the typing judgment $\emptyset \mid \emptyset \vdash \mathit{unbox}\ v : \tau$, as in the rule (ECSTG).

There is a separate cast rule for each kind of target type. The rule (ECSTG) handles the case of casting to a ground type. The cast is removed provided the run-time type exactly matches the target type. The rule (ECSTF) handles the case of casting to a function type. If the run-time type is consistent with the target type, the cast is removed and the inner value is wrapped inside a new function that inserts casts to produce a well-typed value of the appropriate type. This rule is inspired by the work on semantic casts [13, 14, 15], though the rule may look slightly different because the casts used in this paper are annotated with the target type only and not also with the source type. The rule (ECSTR) handles the case of casting to a reference type. The run-time type must exactly match the target type. The rule (ECSTU) handles the case of casting to ? and ensures that nested casts are collapsed to a single cast. The rule (ECSTE) handles the case when the run-time type is not consistent with the target type and produces a *CastError*.

Because the target types of casts are static, the cast form could be replaced by a cast for each type, acting as injection to ? and projection to ground and function types. However, this would complicate the rules, especially the rule for casting to a function type.

The rule (EKILL) terminates evaluation when the derivation depth counter reaches zero.

5.3 Examples

Consider once again the following program and assume the succ constant has the type **number** → **number**.

```
((λ (x) (succ x)) #t)
```

The cast insertion judgement transforms this term into the following term.

```
((λ (x : ?) (succ ⟨number⟩x)) ⟨?⟩#t)
```

Evaluation then proceeds, applying the function to its argument, substituting ⟨?⟩#t for x.

```
(succ ⟨number⟩⟨?⟩#t)
```

The type of #t is **boolean**, which is not consistent with **number**, so the rule (ECSTE) applies and the result is a cast error.

```
CastError
```

Next, we look at an example that uses first-class functions.

```
((λ (f : ? → number) (f 1))
 (λ (x : number) (succ x)))
```

Cast insertion results in the following program.

```
((λ (f : ? → number) (f ⟨?⟩1))
 ⟨? → number⟩(λ (x : number) (succ x)))
```

We apply the cast to the function, creating a wrapper function.

```
((λ (f : ? → number) (f ⟨?⟩1))
 (λ (z : ?) ⟨number⟩((λ (x : number) (succ x)) ⟨number⟩z)))
```

Function application results in the following

Figure 7. Evaluation

$$\boxed{e|\mu \hookrightarrow_n r|\mu}$$

Casting

$$\text{(ECstG)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \gamma}{\langle\gamma\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{unbox}\ v \mid \mu'}$$

$$\text{(ECstF)}\quad \frac{\begin{array}{c} e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \tau \to \tau' \\ \tau \to \tau' \sim \sigma \to \sigma' \qquad z = \mathit{max} v + 1 \end{array}}{\langle\sigma \to \sigma'\rangle\ e \mid \mu \hookrightarrow_{n+1} \lambda\, z{:}\sigma.\ (\langle\sigma'\rangle\ (\mathit{unbox}\ v\ (\langle\tau\rangle\ z))) \mid \mu'}$$

$$\text{(ECstR)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \mathit{ref}\ \tau}{\langle \mathit{ref}\ \tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{unbox}\ v \mid \mu'}$$

$$\text{(ECstU)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu'}{\langle ?\rangle\ e \mid \mu \hookrightarrow_{n+1} \langle ?\rangle\ \mathit{unbox}\ v \mid \mu'}$$

Functions and constants

$$\text{(ELam)}\quad \frac{0 < n}{\lambda\, x{:}\tau.\ e \mid \mu \hookrightarrow_n \lambda\, x{:}\tau.\ e \mid \mu}$$

$$\text{(EApp)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n \lambda\, x{:}\tau.\ e_3 \mid \mu_2 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x{:=}v_2]e_3 \mid \mu_3 \hookrightarrow_n v_3 \mid \mu_4 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} v_3 \mid \mu_4}$$

$$\text{(EConst)}\quad \frac{0 < n}{c \mid \mu \hookrightarrow_n c \mid \mu}$$

$$\text{(EDelta)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n c_1 \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n c_2 \mid \mu_3}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \delta\ c_1\ c_2 \mid \mu_3}$$

References

$$\text{(ERef)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad l \notin \mathit{dom}\ \mu'}{\mathit{ref}\ e \mid \mu \hookrightarrow_{n+1} l \mid \mu'(l \mapsto v)}$$

$$\text{(EDeref)}\quad \frac{e \mid \mu \hookrightarrow_n l \mid \mu' \qquad \mu'\, l = \lfloor v \rfloor}{!e \mid \mu \hookrightarrow_{n+1} v \mid \mu'}$$

$$\text{(EAssign)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n l \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n v \mid \mu_3}{e_1 \leftarrow e_2 \mid \mu_1 \hookrightarrow_{n+1} l \mid \mu_3(l \mapsto v)}$$

$$\text{(ELoc)}\quad \frac{0 < n}{l \mid \mu \hookrightarrow_n l \mid \mu}$$

Figure 8. Evaluation (Errors)

$$\text{(ECstE)}\quad \frac{\begin{array}{c} e \mid \mu \hookrightarrow_n v \mid \mu' \\ \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \sigma \qquad (\sigma, \tau) \notin \mathit{op} \sim \end{array}}{\langle\tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{CastError} \mid \mu'}$$

$$\text{(EKill)}\quad e \mid \mu \hookrightarrow_0 \mathit{KillError} \mid \mu$$

$$\text{(EVarT)}\quad \frac{0 < n}{x \mid \mu \hookrightarrow_n \mathit{TypeError} \mid \mu}$$

$$\text{(EAppT)}\quad \frac{e_1 \mid \mu \hookrightarrow_n v_1 \mid \mu' \qquad v_1 \notin \mathit{FunVal}}{e_1\ e_2 \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

$$\text{(ECstP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{\langle\tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAppP1)}\quad \frac{e_1 \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{e_1\ e_2 \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAppP2)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n v_1 \mid \mu_2 \\ v_1 \in \mathit{FunVal} \qquad e_2 \mid \mu_2 \hookrightarrow_n \varepsilon \mid \mu_3 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_3}$$

$$\text{(EAppP3)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n \lambda\, x{:}\tau.\ e_3 \mid \mu_2 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x{:=}v_2]e_3 \mid \mu_3 \hookrightarrow_n \varepsilon \mid \mu_4 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_4}$$

$$\text{(ERefP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{\mathit{ref}\ e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EDerefP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{!e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAssignP1)}\quad \frac{e_1 \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{e_1 \leftarrow e_2 \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAssignP2)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n l \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n \varepsilon \mid \mu_3}{e_1 \leftarrow e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_3}$$

$$\text{(EDerefT)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \nexists l.\ v = l}{!e \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

$$\text{(EAssignT)}\quad \frac{e_1 \mid \mu \hookrightarrow_n v \mid \mu' \qquad \nexists l.\ v = l}{e_1 \leftarrow e_2 \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

((λ (z : ?) ⟨number⟩((λ (x : number) (succ x)) ⟨number⟩z)) ⟨?⟩1)

and then another function application gives us

⟨number⟩((λ (x : number) (succ x)) ⟨number⟩⟨?⟩1)

We then apply the cast rule for ground types (ECSTG).

⟨number⟩((λ (x : number) (succ x)) 1)

followed by another function application:

⟨number⟩(succ 1)

Then by (EDELTA) we have

⟨number⟩2

and by (ECSTG) we finally have the result

2

5.4 Type Safety

Towards proving type safety we prove the usual lemmas. First, environment expansion and contraction does not change typing derivations. Also, changing the store typing environment does not change the typing derivations as long as the new store typing agrees with the old one. The function *Vars* returns the free and bound variables of an expression.

Lemma 6 (Environment Expansion and Contraction)**.**

- If $\Gamma \mid \Sigma \vdash e : \tau$ and $x \notin \mathit{Vars}\ e$ then $\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau$.
- If $\Gamma(y \mapsto \nu) \mid \Sigma \vdash e : \tau$ and $y \notin \mathit{Vars}\ e$ then $\Gamma \mid \Sigma \vdash e : \tau$.
- If $\Gamma \mid \Sigma \vdash e : \tau$ and $\bigwedge l$. If $l \in \mathit{dom}\ \Sigma$ then $\Sigma' l = \Sigma l$. then $\Gamma \mid \Sigma' \vdash e : \tau$.

Proof Sketch. These properties are proved by induction on the typing derivation. □

Also, substitution does not change the type of an expression.

Lemma 7 (Substitution preserves typing)**.** If $\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e' : \sigma$ then $\Gamma \mid \Sigma \vdash [x{:=}e']e : \tau$.

Proof Sketch. The proof is by strong induction on the size of the expression e, using the inversion and environment expansion lemmas. □

Definition 1. *The store typing judgment, written $\Gamma|\Sigma \models \mu$, holds when the domains of Σ and μ are equal and when for every location l in the domain of Σ there exists a type τ such that $\Gamma|\Sigma \vdash \mu(l) : \tau$.*

Next we prove that n-depth evaluation for the intermediate language $\lambda_{\rightarrow}^{\langle\tau\rangle}$ is sound. Informally, this lemma says that evaluation produces either a value of the appropriate type, a cast error, or *KillError* (because evaluation is cut short), but never a type error. The placement of $e \mid \mu \hookrightarrow_n r \mid \mu'$ in the conclusion of the lemma proves that our evaluation rules are complete, analogous to a progress lemma for small-step semantics. This placement would normally be a naive mistake because not all programs terminate. However, by using n-depth evaluation, we can construct a judgment regardless of whether the program is non-terminating because evaluation is always cut short if the derivation depth exceeds n. But does this lemma handle all terminating programs? The lemma is (implicitly) universally quantified over the evaluation depth n. For every program that terminates there is a depth that will allow it to terminate, and this lemma will hold for that depth. Thus, this lemma applies to all terminating programs and does not apply to non-terminating program, as we intend. We learned of this technique from Ernst, Ostermann, and Cook [11], but its origins go back at least to Volpano and Smith [41].

Lemma 8 (Soundness of evaluation)**.** If $\emptyset \mid \Sigma \vdash e : \tau \wedge \emptyset \mid \Sigma \models \mu$ then $\exists r\ \mu'\ \Sigma'.\ e \mid \mu \hookrightarrow_n r \mid \mu' \wedge \emptyset \mid \Sigma' \models \mu' \wedge (\forall l.\ l \in \mathit{dom}\ \Sigma \longrightarrow \Sigma' l = \Sigma l) \wedge ((\exists v.\ r = v \wedge v \in \mathbb{V} \wedge \emptyset \mid \Sigma' \vdash v : \tau) \vee r = \mathit{CastError} \vee r = \mathit{KillError})$.

Proof. The proof is by strong induction on the evaluation depth. We then perform case analysis on the final step of the typing judgment. The case for function application uses the substitution lemma and the case for casts uses environment expansion. The cases for references and assign use the lemma for changing the store typing. The inversion lemmas are used throughout. □

Theorem 2 (Type safety)**.** If $\emptyset \vdash e \Rightarrow e' : \tau$ then $\exists r\ \mu\ \Sigma.\ e' \mid \emptyset \hookrightarrow_n r \mid \mu \wedge ((\exists v.\ r = v \wedge v \in \mathbb{V} \wedge \emptyset \mid \Sigma \vdash v : \tau) \vee r = \mathit{CastError} \vee r = \mathit{KillError})$.

Proof. Apply Lemma 3 and then Lemma 8. □

6. Relation to Dynamic of Abadi et al.

We defined the semantics for $\lambda_{\rightarrow}^{?}$ with a translation to $\lambda_{\rightarrow}^{\langle\tau\rangle}$, a language with explicit casts. Perhaps a more obvious choice for intermediate language would be the pre-existing language of explicit casts of Abadi et. all [1]. However, there does not seem to be a straightforward translation from $\lambda_{\rightarrow}^{\langle\tau\rangle}$ to their language. Consider the evaluation rule (ECSTF) and how that functionality might be implemented in terms of typecase. The parameter z must be cast to τ, which is not known statically but only dynamically. To implement this cast we would need to dispatch based on τ, perhaps with a typecase. However, typecase must be applied to a value, and there is no way for us to obtain a value of type τ from a value of type $\tau \rightarrow \tau'$. Quoting from [1]:

> Neither tostring nor typetostring quite does its job: for example, when tostring gets to a function, it stops without giving any more information about the function. It can do no better, given the mechanisms we have described, since there is no effective way to get from a function value to an element of its domain or codomain.

Of course, if their language were to be extended with a construct for performing case analysis on types, such as the typerec of Harper and Morrisett [19], it would be straightforward to implement the appropriate casting behavior.

7. Related Work

Several programming languages provide gradual typing to some degree, such as Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. This paper formalizes a type system that provides a theoretical foundation for these languages.

Common LISP [23] and Dylan [12, 37] include optional type annotations, but the annotations are not used for type checking, they are used to improve performance.

Cartwright and Fagan's Soft Typing [7] improves the performance of dynamically typed languages by inferring types and removing the associated run-time dispatching. They do not focus on statically

catching type errors, as we do here, and do not study a source language with optional type annotations.

Anderson and Drossopoulou formalize BabyJ [2], an object-oriented language inspired by JavaScript. BabyJ has a nominal type system, so types are class names and the permissive type $*$. In the type rules for BabyJ, whenever equality on types would normally be used, they instead use the relation $\tau_1 \approx \tau_2$ which holds whenever τ_1 and τ_2 are the same name, or when at least one of them is the permissive type $*$. Our unknown type ? is similar to the permissive type $*$, however, the setting of our work is a structural type system and our type compatibility relation $\sim$ takes into account function types.

Riely and Hennessy [35] define a partial type system for Dπ, a distributed π-calculus. Their system allows some locations to be untyped and assigns such locations the type lbad. Their type system, like Quasi-Static Typing, relies on subtyping, however they treat lbad as "bottom", which allows objects of type lbad to be implicitly coercible to any other type.

Gradual typing is syntactically similar to type inferencing [9, 21, 27]: both approaches allow type annotations to be omitted. However, with type inference, the type system tries to reconstruct what the type annotations should be, and if it cannot, rejects the program. In contrast, a gradual type system accepts that it does not know certain types and inserts run-time casts.

Henglein [20] presents a translation from untyped λ-terms to a coercion calculus with explicit casts. These casts make explicit the tagging, untagging, and tag-checking operations that occur during the execution of a language with latent (dynamic) typing. Henglein's coercion calculus seems to be closely related to our $\lambda^{\langle\tau\rangle}_{\rightarrow}$ but we have not yet formalized the relation. Henglein does not study a source language with partially typed terms with a static type system, as we do here. Instead, his source language is a dynamically typed language.

Bracha [4] defines *optional type systems* as type systems that do not affect the semantics of the language and where type annotations are optional. Bracha cites Strongtalk [5] as an example of an optional type system, however, that work does not define a formal type system or describe how omitted type annotations are treated.

Ou et. all. [31] define a language that combines standard static typing with more powerful dependent typing. Implicit coercions are allowed to and from dependent types and run-time checks are inserted. This combination of a weaker and a stronger type system is analogous to the combination of dynamic typing and static typing presented in this paper.

Flanagan [15] introduces Hybrid Type Checking, which combines standard static typing with refinement types, where the refinements may express arbitrary predicates. The type system tries to satisfy the predicates using automated theorem proving, but when no conclusive answer is given, the system inserts run-time checks. This work is also analogous to ours in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. One notable difference between our system and Flanagan's is that his is based on subtyping whereas ours is based on the consistency relation.

Gronski, Knowles, Tomb, Freund, and Flanagan [18] developed the Sage language which provides Hybrid Type Checking and also a Dynamic type with implicit (run-time checked) down-casts. Surprisingly, the Sage type system does not allow implicit down-casts from Dynamic, whereas the Sage type checking (and compilation) algorithm does allow implicit down-casts. It may be that the given type system was intended to characterize the output of compilation (though it is missing a rule for cast), but then a type system for the source language remains to be defined. The Sage technical report [18] does not include a result such as Theorem 1 of this paper to show that the type system catches all type errors for fully annotated programs, which is a tricky property to achieve in the presence of a Dynamic type with implicit down-casts.

There are many interesting issues regarding efficient representations for values in a language that mixes static and dynamic typing. The issues are the same as for parametric polymorphism (dynamic typing is just a different kind of polymorphism). Leroy [25] discusses the use of mixing boxed and unboxed representations and such an approach is also possible for our gradual type system. Shao [38] further improves on Leroy's mixed approach by showing how it can be combined with the type-passing approach of Harper and Morrisett [19] and thereby provide support for recursive and mutable types.

8. Conclusion

The debate between dynamic and static typing has continued for several decades, with good reason. There are convincing arguments for both sides. Dynamic typing is better suited than static for prototyping, scripting, and gluing components, whereas static typing is better suited for algorithms, data-structures, and systems programming. It is common practice for programmers to start development of a program in a dynamic language and then translate to a static language midway through development. However, static and dynamic languages are often radically different, making this translation difficult and error prone. Ideally, migrating between dynamic to static could take place gradually and while staying within the same language.

In this paper we present the formal definition of the language $\lambda^{?}_{\rightarrow}$, including its static and dynamic semantics. This language captures the key ingredients for implementing gradual typing in functional languages. The language $\lambda^{?}_{\rightarrow}$ provides the flexibility of dynamically typed languages when type annotations are omitted by the programmer and provides the benefits of static checking when all function parameters are annotated, including the safety guarantees (Theorem 1) and the time and space efficiency (Lemma 5). Furthermore, the cost of dynamism is "pay-as-you-go", so partially annotated programs enjoy the benefits of static typing to the degree that they are annotated. We prove type safety for $\lambda^{?}_{\rightarrow}$ (Theorem 2); the type system prevents type violations from occurring at run-time, either by catching the errors statically or by catching them dynamically with a cast exception. The type system and run-time semantics of $\lambda^{?}_{\rightarrow}$ is relatively straightforward, so it is suitable for practical languages.

As future work, we intend to investigate the interaction between our gradual type system and types such as lists, arrays, algebraic data types, and implicit coercions between types, such as the types in Scheme's numerical tower. We also plan to investigate the interaction between gradual typing and parametric polymorphism [16, 34] and Hindley-Milner inference [9, 21, 27]. We have implemented and tested an interpreter for the $\lambda^{?}_{\rightarrow}$ calculus. As future work we intend to incorporate gradual typing as presented here into a mainstream dynamically typed programming language and perform studies to evaluate whether gradual typing can benefit programmer productivity.

Acknowledgments

We thank the anonymous reviewers for their suggestions. We thank Emir Pasalic the members of the Resource Aware Programming Laboratory for reading drafts and suggesting improvements. This work was supported by NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced

Languages Techniques for Device Drivers, and NSF SOD-0439017 Synthesizing Device Drivers.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

[3] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

[4] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[5] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

[6] Y. Bres, B. P. Serpette, and M. Serrano. Compiling scheme programs to .NET common intermediate language. In *2nd International Workshop on .NET Technologies*, Pilzen, Czech Republic, May 2004.

[7] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[8] C. Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[10] R. B. de Oliveira. The Boo programming language. http://boo.codehaus.org, 2005.

[11] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[12] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.

[14] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.

[15] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[16] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[17] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

[18] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. Technical report, University of California, Santa Cruz, 2006.

[19] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.

[20] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[21] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans AMS*, 146:29–60, 1969.

[22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[23] G. L. S. Jr. An overview of COMMON LISP. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.

[24] R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[25] X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.

[26] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[28] T. Nipkow. Structured proofs in Isar/HOL. In *TYPES*, number 2646 in LNCS, 2002.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] A. Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994.

[31] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types (extended abstract). In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.

[32] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[33] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[34] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[35] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–104, New York, NY, USA, 1999. ACM Press.

[36] M. Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

[37] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[38] Z. Shao. Flexible representation analysis. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 85–98, New York, NY, USA, 1997. ACM Press.

[39] J. Siek and W. Taha. Gradual typing: Isabelle/isar formalization. Technical Report TR06-874, Rice University, Houston, Texas, 2006.

[40] S. Thatte. Quasi-static typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.

[41] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW'97: 10th Computer Security Foundations Workshop*, volume 00, page 156, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[42] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, April 2004.

SAGE: Hybrid Checking for Flexible Specifications

Jessica Gronski[†] Kenneth Knowles[†] Aaron Tomb[†] Stephen N. Freund[‡] Cormac Flanagan[†]

[†]University of California, Santa Cruz [‡] Williams College

Abstract

Software systems typically contain large APIs that are informally specified and hence easily misused. This paper presents the SAGE programming language, which is designed to enforce precise interface specifications in a flexible manner. The SAGE type system uses a synthesis of the type `Dynamic`, first-class types, and arbitrary refinement types. Since type checking for this expressive language is not statically decidable, SAGE uses *hybrid type checking*, which extends static type checking with dynamic contract checking, automatic theorem proving, and a database of refuted subtype judgments.

1. Introduction

Constructing a large, reliable software system is extremely challenging, due to the difficulty of understanding the system in its entirety. A necessary strategy for controlling this conceptual complexity is to divide the system into modules that communicate via clearly specified interfaces.

The precision of these interface specifications may naturally and appropriately evolve during the course of software development. To illustrate this potential variation, consider the following specifications for the argument to a function `invertMatrix`:

1. The argument can be any (dynamically-typed) value.
2. The argument must be an array of arrays of numbers.
3. The argument must be a *matrix*, that is, a rectangular (non-ragged) array of arrays of numbers.
4. The argument must be a square matrix.
5. The argument must be a square matrix that satisfies the predicate `isInvertible`.

All of these specifications are valid constraints on the argument to `invertMatrix`, although some are obviously more precise than others. Different specifications may be appropriate at different stages of the development process. Simpler specifications facilitate rapid prototyping, whereas more precise specifications provide more correctness guarantees and better documentation.

Traditional statically-typed languages, such as Java, C#, and ML, primarily support the second of these specifications. Dynamically-typed languages such as Scheme primarily support the first specification. Contracts [32, 13, 26, 21, 24, 28, 37, 25, 12, 8] provide a means to document and enforce all of these specifications, but violations are only detected dynamically, resulting in incomplete and late (possibly post-deployment) detection of defects. This paper presents the SAGE programming language and type system, which is designed to support and enforce a wide range of specification methodologies. SAGE verifies correctness properties and detects defects via static checking wherever possible. However, SAGE can also enforce specifications dynamically, when necessary.

On a technical level, the SAGE type system can be viewed as a synthesis of three concepts: the type `Dynamic`; arbitrary refinement types; and first-class types. These features add expressive power in three orthogonal directions, yet they all cooperate neatly within SAGE's hybrid static/dynamic checking framework.

Type `Dynamic`. The type `Dynamic` [23, 1, 39] enables SAGE to support dynamically-typed programming. `Dynamic` is a supertype of all types; any value can be upcast to type `Dynamic`, and a value of declared type `Dynamic` can be implicitly downcast (via a run-time check) to a more precise type. Such downcasts are implicitly inserted when necessary, such as when the operation `add1` (which expects an `Int`) is applied to a variable of type `Dynamic`. Thus, declaring variables to have type `Dynamic` (which is the default if type annotations are omitted) leads to a dynamically-typed, Scheme-like style of programming.

These dynamically-typed programs can later be annotated with traditional type specifications like `Int` and `Bool`. One nice aspect of our system is that the programmer need not fully annotate the program with types in order to reap some benefit. Types enable SAGE to check more properties statically, but it is still able to fall back to dynamic checking whenever the type `Dynamic` is encountered.

Refinement Types. For increased precision, SAGE also supports *refinement types*. For example, the following code snippet defines the type of integers in the range from `lo` (inclusive) to `hi` (exclusive):

```
{ x:Int | lo <= x && x < hi }
```

SAGE extends prior work on decidable refinement types [44, 43, 18, 30, 35] to support arbitrary executable refinement predicates — any boolean expression can be used as a refinement predicate.

First-Class Types. Finally, SAGE elevates types to be first-class values, in the tradition of Pure Type Systems [5]. Thus, types can be returned from functions, which permits function abstractions to abstract over types as well as terms. For example, the following function `Range` takes two integers and returns the *type* of integers within that range:

```
let Range (lo:Int) (hi:Int) : *
    = { x:Int | lo <= x && x < hi };
```

Here, `*` is the type of types and indicates that `Range` returns a type. Similarly, we can pass types to functions, as in the following polymorphic identity function:

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

```
let id (T:*) (x:T) : T = x;
```

The traditional limitation of both first-class types and unrestricted refinement types is that they are not statically decidable. SAGE circumvents this difficulty by replacing *static* type checking with *hybrid* type checking [14]. SAGE checks correctness properties and detects defects statically, whenever possible. However, it resorts to dynamic checking for particularly complicated specifications. The overall result is that precise specifications can be enforced, with most errors detected at compile time, and violations of some complicated specifications detected at run time.

1.1 Hybrid Type Checking

We briefly illustrate the key idea of hybrid type checking by considering the function application

$$(\texttt{factorial}\ t)$$

Suppose `factorial` has type `Pos` $\rightarrow$ `Pos`, where `Pos` $= \{\texttt{x} : \texttt{Int} \mid \texttt{x} > \texttt{0}\}$ is the type of positive integers, and that t has some type T. If the type checker can prove (or refute) that $T <:$ `Pos`, then this application is well-typed (or ill-typed, respectively). However, if T is a complex refinement type or a complex type-producing computation, the SAGE compiler may be unable to either prove or refute that $T <:$ `Pos` because subtyping is undecidable.

In this situation, *statically accepting* the application may result in the specification of `factorial` being violated at run time, which is clearly unacceptable. Alternatively, *statically rejecting* such programs would cause the compiler to reject some well-typed programs, as in the Cayenne compiler [4]. Our previous experience with ESC/Java [17] indicates that this is too brittle in practice.

One solution to this dilemma is to require that the programmer provide a proof that $T <:$ `Pos` (see *e.g.* [7, 36, 10]). While this approach is promising for critical software, it may be somewhat heavyweight for widespread use.

Instead, SAGE enforces the specification of `factorial` dynamically, by inserting the following *type cast* to ensure at run time that the result of t is a positive integer:

$$\texttt{factorial}\ (\langle\texttt{Pos}\rangle\ t)$$

This approach works regardless of whether T is the type `Dynamic`, a complex refinement type, or a complex type-producing computation.

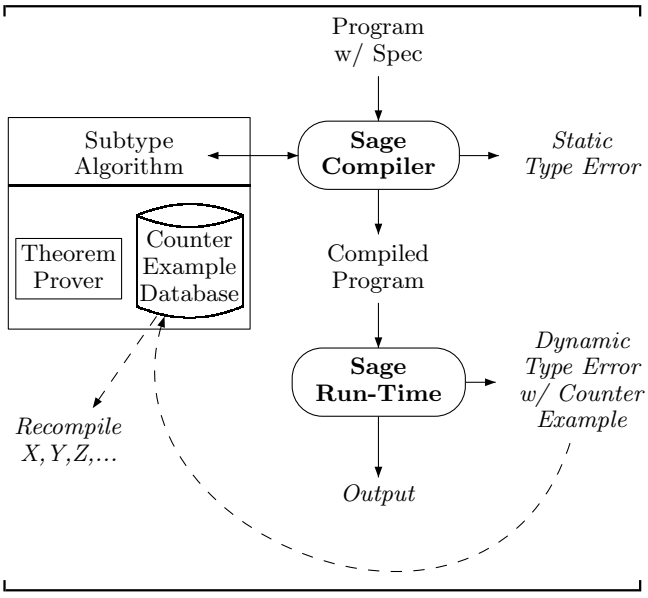
Hybrid combinations of static and dynamic checking are not new. For example, many existing language implementations enforce type safety for arrays using both static type checks and dynamic bounds checks. Hybrid type checking extends this approach to enforce user-defined specifications. Hybrid type checking also extends ideas from soft typing [29, 42, 3], to detect type errors at compile time, in the spirit of static type systems.

Prior work explored hybrid type checking in an idealized setting, that of the simply-typed lambda-calculus with refinements only on the base types `Int` and `Bool` [14]. This paper adapts hybrid type checking to the more technically challenging domain of a rich language that includes all of the features described above. We also provide an implementation and experimental validation of this approach.

1.2 The SAGE Compilation Algorithm

The overall architecture of the SAGE compiler is shown in Figure 1. A key component of the SAGE compiler is its subtype algorithm, which attempts to prove or disprove a subtype relationship $S <: T$ (in the context of an environment E). To obtain adequate precision, the subtype algorithm incorporates the following two modules:

Figure 1: Sage Architecture



Theorem Prover. Testing subtyping between refinement types reduces to testing implication between refinement predicates. In order to reason about these predicates, the subtype algorithm translates each implication between SAGE predicates into a validity test of a logical formula, which can be passed to an automated theorem prover (currently Simplify [11]).

Counter-Example Database. If a compiler-inserted cast from type S to type T fails at run time, SAGE stores in a counter-example database the fact that S is not a subtype of T. The subtype algorithm consults this database during compilation and will subsequently reject any program that relies on S being a subtype of T. Thus, dynamic type errors actually improve the ability of the SAGE compiler to detect type errors statically.

Moreover, when a compiler-inserted cast fails, SAGE will report a list of previously-compiled programs that contain the same (or a sufficiently similar) cast, since these programs may also fail at run time. Thus, the counter-example database functions somewhat like a regression test suite, in that it can detect errors in previously compiled programs.

Over time, we predict that the database will grow to be a valuable repository of common but invalid subtype relationships, leading to further improvements in the checker's precision and less reliance on compiler-inserted casts.

The combination of these features yields a subtype algorithm that is quite precise — the number of compiler-inserted casts is very small or zero on all of our benchmarks. Dynamic checks are only necessary for a few particularly complicated cases.

1.3 Contributions

The primary contributions of this paper are as follows:

- We present the SAGE programming language, which supports flexible specifications in a syntactically lightweight manner by combining arbitrary refinement types with first-class types and the type `Dynamic`.

- We present a hybrid type checking algorithm for SAGE that circumvents the decidability limitations of this expressive type language. This type checker accepts all (arbitrarily complicated) well-typed programs, and enforces all interface specifications, either statically or dynamically. The checker integrates compile-time evaluation, theorem proving, and a database of failed type casts.
- We provide experimental results for a prototype implementation of SAGE. These results show that SAGE verifies the vast majority of specifications in our benchmark programs statically.

The following section illustrates the SAGE language through a series of examples. Sections 3 and 4 define the syntax, semantics, and type system for SAGE. Section 5 presents a hybrid compilation algorithm for the language. Sections 6 and 7 describe our implementation and experimental results. Sections 8 and 9 discuss related work and future plans.

2. Motivating Examples

We introduce SAGE through several examples illustrating key features of the language, including refinement types, dependent function types, datatypes, and recursive types. We focus primarily on programs with fairly complete specifications to highlight these features. Programs could rely more on the type `Dynamic` than these, albeit with fewer static guarantees.

2.1 Binary Search Trees

We begin with the commonly-studied example of binary search trees, whose SAGE implementation is shown in Figure 2. The variable `Range` is of type `Int` → `Int` → `*`, where `*` is the type of types. Given two integers `lo` and `hi`, the application `Range lo hi` returns the following refinement type describing integers in the range [`lo`, `hi`):

```
{x:Int | lo <= x && x < hi }
```

A binary search tree `(BST lo hi)` is an ordered tree containing integers in the range [`lo`, `hi`). A tree may either be `Empty`, or a `Node` containing an integer `v` ∈ [`lo`, `hi`) and two subtrees containing integers in the ranges [`lo`, `v`) and [`v`, `hi`), respectively. Thus, the type of binary search trees explicates the requirement that these trees must be ordered.

The function `search` takes as arguments two integers `lo` and `hi`, a binary search tree of type `(BST lo hi)`, and an integer `x` in the range [`lo`, `hi`). Note that SAGE supports dependent function types, and so the type of the third argument to `search` can depend on the values of the first and second arguments. The function `search` then checks if `x` is in the tree. The function `insert` takes similar arguments and extends the given tree with the integer `x`.

The SAGE compiler uses an automatic theorem prover to statically verify that the specified ordering invariants on binary search trees are satisfied by these two functions. Thus, no run-time checking is required for this example.

The precise type specifications enable SAGE to detect various common programming errors. For example, suppose we inadvertently used the wrong conditional test:

```
24:       if x <= v
```

For this (incorrect and ill-typed) program, the SAGE compiler will report that the specification for `insert` is violated by the first recursive call:

```
line 25: x does not have type (Range lo v)
```

Figure 2: Binary Search Trees

```
 1: let Range (lo:Int) (hi:Int) : * =
 2:   {x:Int | lo <= x && x < hi };
 3:
 4: datatype BST (lo:Int) (hi:Int) =
 5:   Empty
 6: | Node of (v:Range lo hi)*(BST lo v)*(BST v hi);
 7:
 8: let rec search (lo:Int) (hi:Int) (t:BST lo hi)
 9:                (x:Range lo hi) : Bool =
10:   case t of
11:     Empty -> false
12:   | Node v l r ->
13:      if x = v then true
14:      else if x < v
15:           then search lo v l x
16:           else search v hi r x;
17:
18: let rec insert (lo:Int) (hi:Int) (t:BST lo hi)
19:                (x:Range lo hi) : (BST lo hi) =
20:   case t of
21:     Empty ->
22:      Node lo hi x (Empty lo x) (Empty x hi)
23:   | Node v l r ->
24:      if x < v
25:      then Node lo hi v (insert lo v l x) r
26:      else Node lo hi v l (insert v hi r x);
```

Similarly, if one of the arguments to the constructor `Node` is incorrect, *e.g.*:

```
26:       else Node lo hi v r (insert v hi r x);
```

the SAGE compiler will report the type error:

```
line 26: r does not have type (BST lo v)
```

Notably, a traditional type system that does not support precise specifications would not detect either of these errors.

Using this `BST` implementation, constructing trees with specific constraints is straightforward (and verifiable). For example, the following code constructs a tree containing only positive numbers:

```
let PosBST : * = BST 1 MAXINT;
let nil : PosBST = Empty 1 MAXINT;
let add (t:PosBST) (x:Range 1 MAXINT) : PosBST =
        insert 1 MAXINT t x;
let find (t:PosBST) (x:Range 1 MAXINT) : Bool =
        search 1 MAXINT t x;

let t : PosBST = add (add (add nil 1) 3) 5;
```

Note that this fully-typed BST implementation interoperates with dynamically-typed client code:

```
let t : Dynamic = (add nil 1) in find t 5;
```

2.2 Regular Expressions

We now consider a more complicated specification. Figure 3 declares the `Regexp` data type and the function `match`, which determines if a string matches a regular expression. The `Regexp` datatype includes constructors to match any single letter (`Alpha`) or any single letter or digit (`AlphaNum`), as well as usual the Kleene closure, concatenation, and choice operators. As an example, the regular expression

Figure 3: Regular Expressions and Names

```
datatype Regexp =
  Alpha
| AlphaNum
| Kleene of Regexp
| Concat of Regexp * Regexp
| Or of Regexp * Regexp
| Empty;

let match (r:Regexp) (s:String) : Bool = ...

let Name = {s:String | match (Kleene AlphaNum) s};
```

"`[a-zA-Z0-9]*`" would be represented in our datatype as `(Kleene AlphaNum)`.

The code then uses `match` to define the type `Name`, which refines the type `String` to allow only alphanumeric strings. We use the type `Name` to enforce an important, security-related interface specification for the following function `authenticate`. This function performs authentication by querying a SQL database (where '^' denotes string concatenation):

```
let authenticate (user:Name) (pass:Name) : Bool =
   let query : String =
     ("SELECT count(*) FROM client WHERE name =" ^
     user ^ " and pwd=" ^ pass) in
   executeSQLquery(query) > 0;
```

This code is prone to security attacks if given specially-crafted non-alphanumeric strings. For example, calling

```
authenticate  "admin --"  ""
```

breaks the authentication mechanism because "`--`" starts a comment in SQL and consequently "comments out" the password part of the query. To prohibit this vulnerability, the type:

```
authenticate : Name → Name → Bool
```

specifies that `authenticate` should be applied only to alphanumeric strings.

Next, consider the following user-interface code:

```
let username : String = readString() in
let password : String = readString() in
authenticate username password;
```

This code is ill-typed, since it passes arbitrary user input of type `String` to `authenticate`. However, proving that this code is ill-typed is quite difficult, since it depends on complex reasoning showing that the user-defined function `match` is not a tautology, and hence that not all `String`s are `Name`s.

In fact, Sage cannot statically verify or refute this code. Instead, it inserts the following casts at the call site to enforce the specification for `authenticate` dynamically:

```
authenticate (⟨Name⟩ username) (⟨Name⟩ password);
```

At run time, these casts check that `username` and `password` are alphanumeric strings satisfying the predicate `match (Kleene AlphaNum)`. If the `username` "`admin --`" is ever entered, the cast `(⟨Name⟩ username)` will fail and halt program execution.

2.3 Counter-Example Database

Somewhat surprisingly, a dynamic cast failure actually strengthens Sage's ability to detect type errors statically. In particular, the string "`admin --`" is a witness proving that not all `String`s are `Name`s, *i.e.*, $E \not\vdash$ `String` <: `Name` (where E is the typing environment for the call to `authenticate`). Rather than discarding this information, and potentially observing the same error on later runs or in different programs, such refuted subtype relationships are stored in a database. If the above code is later re-compiled, the Sage compiler will discover upon consulting this database that `String` is not a subtype of `Name`, and it will statically reject the call to `authenticate` as ill-typed.

Additionally, the database stores a list of other programs previously compiled under the assumption that `String` may be a subtype of `Name`. These programs may also fail at run time and Sage will also report that they must be recompiled or modified to be accepted by the more-informed checker. It remains to be seen how to best incorporate this feature into a development process.

2.4 Printf

As a final example, we examine the `printf` function. The number and type of the expected arguments to `printf` depends in subtle ways on the format string (the first argument). In Sage, we can assign to `printf` the precise type:

```
printf : (format:String) -> (Printf_Args format)
```

where the user-defined function

```
Printf_Args : String -> *
```

returns the `printf` argument types for the given format string. For example, `(Printf_Args "%d%d")` evaluates to the type `Int` → `Int` → `Unit`. Calls to `printf` are assigned precise types, such as:

```
printf "%d%d" : Int -> Int -> Unit
```

since this term has type `(Printf_Args "%d%d")`, which in turn evaluates to `Int` → `Int` → `Unit`.

Thus, the Sage language is sufficiently expressive to need no special support for accommodating `printf` and catching errors in `printf` clients statically. In contrast, other language implementations require special rules in the compiler or run time to ensure the type safety of calls to `printf`. For example, Scheme [40] and GHC [19, 38] leave all type checking of arguments to the run time. OCaml [27], on the other hand, performs static checking, but it requires the format string to be constant.

Sage can statically check many uses of `printf` with non-constant format strings, as illustrated by the following example:

```
let repeat (s:String) (n:Int) : String =
  if (n = 0) then "" else (s ^ (repeat s (n-1)));

// checked statically:
printf (repeat "%d" 2) 1 2;
```

The Sage compiler infers that `printf (repeat "%d" 2)` has type `Printf_Args (repeat "%d" 2)`, which evaluates (at compile-time) to `Int` → `Int` → `Unit`, and hence this call is well-typed. Conversely, the compiler would statically reject the following ill-typed call:

```
// compile-time error:
printf (repeat "%d" 2) 1 false;
```

For efficiency, and to avoid non-termination, the compiler performs only a bounded number of evaluation steps before resorting to dynamic checking. Thus, the following call requires a run-time check:

Figure 4: Syntax, Constants, and Shorthands

Syntax:

$s, t, S, T ::=$	*Terms:*
x	variable
c	constant
$\texttt{let}\ x = t : S\ \texttt{in}\ t$	binding
$\lambda x{:}S.\ t$	abstraction
$t\ t$	application
$x{:}S \to T$	function type

Constants:

$$
\begin{array}{rcl}
\texttt{*} & : & \texttt{*} \\
\texttt{Unit} & : & \texttt{*} \\
\texttt{Bool} & : & \texttt{*} \\
\texttt{Int} & : & \texttt{*} \\
\texttt{Dynamic} & : & \texttt{*} \\
\texttt{Refine} & : & X{:}\texttt{*} \to (X \to \texttt{Bool}) \to \texttt{*} \\
 & & \\
\texttt{unit} & : & \texttt{Unit} \\
\texttt{true} & : & \{b{:}\texttt{Bool} \mid b\} \\
\texttt{false} & : & \{b{:}\texttt{Bool} \mid \texttt{not}\ b\} \\
\texttt{not} & : & b{:}\texttt{Bool} \to \{b'{:}\texttt{Bool} \mid b' = \texttt{not}\ b\} \\
n & : & \{m{:}\texttt{Int} \mid m = n\} \\
+ & : & n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
= & : & x{:}\texttt{Dynamic} \to y{:}\texttt{Dynamic} \\
 & & \to \{b{:}\texttt{Bool} \mid b = (x = y)\} \\
 & & \\
\texttt{if} & : & X{:}\texttt{*} \to p{:}\texttt{Bool} \\
 & & \to (\{d{:}\texttt{Unit} \mid p\} \to X) \\
 & & \to (\{d{:}\texttt{Unit} \mid \texttt{not}\ p\} \to X) \\
 & & \to X \\
 & & \\
\texttt{fix} & : & X{:}\texttt{*} \to (X \to X) \to X \\
\texttt{cast} & : & X{:}\texttt{*} \to \texttt{Dynamic} \to X
\end{array}
$$

Shorthands:

$$
\begin{array}{rcll}
S \to T & = & x{:}S \to T & x \notin FV(T) \\
\langle T \rangle & = & \texttt{cast}\ T & \\
\{x{:}T \mid t\} & = & \texttt{Refine}\ T\ (\lambda x{:}T.\ t) & \\
\texttt{if}_T\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & = & & \\
\multicolumn{4}{l}{\quad \texttt{if}\ T\ t_1\ (\lambda x{:}\{d{:}\texttt{Unit} \mid t\}.\ t_2)\ (\lambda x{:}\{d{:}\texttt{Unit} \mid \texttt{not}\ t\}.\ t_3)}
\end{array}
$$

```
// run-time error:
printf (repeat "%d" 20) 1 2 ... 19 false;
```

As expected, the inserted dynamic cast catches the error.

Our current SAGE implementation is not yet able to statically verify that the implementation of `printf` matches its specification (`format:String` → (`Printf_Args format`)). As a result, the compiler inserts a single dynamic type cast into the `printf` implementation. This example illustrates the flexibility of hybrid checking — the `printf` specification is enforced dynamically on the `printf` implementation, but also enforced (primarily) statically on client code. We revisit this example in Section 5.3 to illustrate SAGE's compilation algorithm.

3. Language

3.1 Syntax and Informal Semantics

SAGE programs are desugared into a small core language, whose syntax and semantics are described in this section. Since types are first class, SAGE merges the syntactic categories of terms and types [5]. The syntax of the resulting type/term language is summarized in Figure 4. We use the following naming convention to distinguish the intended use of meta-variables: x, y, z range over regular program variables; X, Y, Z range over type variables; s, t range over regular program terms; and S, T range over types.

The core SAGE language includes variables, constants, functions, function applications, and let expressions. The language also includes dependent function types, for which we use Cayenne's [4] syntax $x : S \to T$ (in preference over the equivalent notation $\Pi x : S.T$). Here, S specifies the function's domain, and the formal parameter x can occur free in the range type T. We use the shorthand $S \to T$ when x does not occur free in T.

The SAGE type system assigns a type to each well-formed term. Since each type is simply a particular kind of term, it is also assigned a type, specifically the type "`*`", which is the type of types [9]. Thus, `Int`, `Bool`, and `Unit` all have type `*`. Also, `*` itself has type `*`.

The unification of types and terms allows us to pass types to and from functions. For example, the following function `UnaryOp` is a type operator that, given a type such as `Int`, returns the type of functions from `Int` to `Int`.

$$\texttt{UnaryOp} \stackrel{\text{def}}{=} \lambda X{:}\texttt{*}.\ (X \to X)$$

Type-valued arguments also support the definition of polymorphic functions, such as `applyTwice`, where the application `applyTwice Int add1` returns a function that adds two to any integer. Thus, polymorphic instantiation is explicit in SAGE.

$$\texttt{applyTwice} \stackrel{\text{def}}{=} \lambda X{:}\texttt{*}.\ \lambda f{:}(\texttt{UnaryOp}\ X).\ \lambda x{:}X.\ f(f(x))$$

The constant `Refine` enables precise refinements of existing types. Suppose $f : T \to \texttt{Bool}$ is some arbitrary predicate over type T. Then the type `Refine` T f denotes the *refinement* of T containing all values of type T that satisfy the predicate f. Following Ou et al. [35], we use the shorthand $\{x{:}T \mid t\}$ to abbreviate `Refine` T $(\lambda x{:}T.\ t)$. Thus, $\{x{:}\texttt{Int} \mid x \geq 0\}$ denotes the type of natural numbers.

We use refinement types to assign precise types to constants. For example, as shown in Figure 4, an integer n has the precise type $\{m{:}\texttt{Int} \mid m = n\}$ denoting the singleton set $\{n\}$. Similarly, the type of the operation + specifies that its result is the sum of its arguments:

$$n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\}$$

The apparent circularity where the type of + is defined in terms of + itself does not cause any difficulties in our technical development, since the semantics of refinement types is defined in terms of the operational semantics.

The type of the primitive `if` is also described via refinements. In particular, the "then" parameter to `if` is a thunk of type $(\{d{:}\texttt{Unit} \mid p\} \to X)$. That thunk can be invoked only if the domain $\{d{:}\texttt{Unit} \mid p\}$ is inhabited, *i.e.*, only if the test expression p evaluates to `true`. Thus the type of `if` precisely specifies its behavior.

The constant `fix` enables the definition of recursive functions and recursive types. For example, the type of integer lists is defined via the least fixpoint operation:

$$\texttt{fix}\ \texttt{*}\ (\lambda L{:}\texttt{*}.\ \texttt{Sum Unit (Pair Int}\ L))$$

which (roughly speaking) returns a type `L` satisfying the equation:

$$L \quad = \quad \texttt{Sum Unit (Pair Int}\ L)$$

Figure 5: Evaluation Rules

Evaluation $\boxed{s \longrightarrow t}$

$$\mathcal{E}[s] \longrightarrow \mathcal{E}[t] \quad \text{if } s \longrightarrow t \qquad \text{[E-Compat]}$$

$$(\lambda x{:}S.\ t)\ v \longrightarrow t[x := v] \qquad \text{[E-App]}$$
$$\texttt{let } x = v : S \texttt{ in } t \longrightarrow t[x := v] \qquad \text{[E-Let]}$$
$$\texttt{not true} \longrightarrow \texttt{false} \qquad \text{[E-Not1]}$$
$$\texttt{not false} \longrightarrow \texttt{true} \qquad \text{[E-Not2]}$$
$$\texttt{if}_T \texttt{ true } v_1\ v_2 \longrightarrow v_1 \texttt{ unit} \qquad \text{[E-If1]}$$
$$\texttt{if}_T \texttt{ false } v_1\ v_2 \longrightarrow v_2 \texttt{ unit} \qquad \text{[E-If2]}$$
$$+\ n_1\ n_2 \longrightarrow n \quad n = (n_1 + n_2) \qquad \text{[E-Add]}$$
$$=\ v_1\ v_2 \longrightarrow c \quad c = (v_1 \equiv v_2) \qquad \text{[E-Eq]}$$

$$\langle \texttt{Bool} \rangle \texttt{ true} \longrightarrow \texttt{true} \qquad \text{[E-Cast-Bool1]}$$
$$\langle \texttt{Bool} \rangle \texttt{ false} \longrightarrow \texttt{false} \qquad \text{[E-Cast-Bool2]}$$
$$\langle \texttt{Unit} \rangle \texttt{ unit} \longrightarrow \texttt{unit} \qquad \text{[E-Cast-Unit]}$$
$$\langle \texttt{Int} \rangle\ n \longrightarrow n \qquad \text{[E-Cast-Int]}$$
$$\langle \texttt{Dynamic} \rangle\ v \longrightarrow v \qquad \text{[E-Cast-Dyn]}$$

$$\langle x{:}S \to T \rangle\ v \longrightarrow \lambda x{:}S.\ \langle T \rangle\ (v\ (\langle D \rangle\ x)) \quad \text{where } D = \mathit{domain}(v) \qquad \text{[E-Cast-Fn]}$$

$$\langle \texttt{Refine } T\ f \rangle\ v \longrightarrow \langle T \rangle\ v \quad \text{if } f\ (\langle T \rangle\ v) \longrightarrow^* \texttt{true} \qquad \text{[E-Refine]}$$

$$\langle * \rangle\ v \longrightarrow v \quad \text{if } v \in \{\texttt{Int}, \texttt{Bool}, \texttt{Unit}, \texttt{Dynamic}, x{:}S \to T, \texttt{fix } * f\} \qquad \text{[E-Cast-Type]}$$

$$\mathcal{S}[\texttt{fix } U\ v] \longrightarrow \mathcal{S}[v\ (\texttt{fix } U\ v)] \qquad \text{[E-Fix]}$$

$$\mathcal{E} ::= \bullet \mid \mathcal{E}\ t \mid v\ \mathcal{E} \qquad \textit{Evaluation Contexts}$$
$$\mathcal{S} ::= \bullet \mid v \mid \langle \bullet \rangle\ v \qquad \textit{Strict Contexts}$$

$u, v, U, V ::=$	*Values*
$\lambda x{:}S.\ t$	abstraction
$x{:}S \to T$	function type
c	constant
$c\ v_1\ \ldots\ v_n$	constant, $0 < n < \mathit{arity}(c)$
$\texttt{Refine } U\ v$	refinement
$\texttt{fix } U\ v$	recursive type

(Here, `Sum` and `Pair` are the usual type constructors for sums and pairs, respectively.)

The Sage language includes two constants that are crucial for enabling hybrid type checking: `Dynamic` and `cast`. The type constant `Dynamic` [1, 39] can be thought of as the most general type. Every value has type `Dynamic`, and casts can be used to convert values from type `Dynamic` to other types (and of course such downcasts may fail if applied to inappropriate values).

The constant `cast` performs dynamic checks or coercions between types. It takes as arguments a type T and a value (of type `Dynamic`), and it attempts to cast that value to type T. We use the shorthand $\langle T \rangle\ t$ to abbreviate `cast` $T\ t$. Thus, for example, the expression

$$\langle \{x{:}\texttt{Int} \mid x \geq 0\} \rangle\ y$$

casts the integer y to the refinement type of natural numbers, and fails if y is negative.

3.2 Operational Semantics

We formalize the execution behavior of Sage programs with the small-step operational semantics shown in Figure 5. Evaluation is performed inside evaluation contexts $\mathcal{E}$. Application, let expressions, and the basic integer and boolean operations behave as expected. Rule [E-Eq] uses syntactic equality ($\equiv$) to test equivalence of all values, including function values[1].

The most interesting reduction rules are those for casts $\langle T \rangle\ v$. Casts to one of the base types `Bool`, `Unit`, or `Int` succeed if the value v is of the appropriate type. Casts to type `Dynamic` always succeed.

Casts to function and refinement types are more complex. First, the following partial function *domain* returns the domain of a function value, and is defined by:

$$\begin{array}{rcl} \mathit{domain} : \mathit{Value} & \to & \mathit{Term} \\ \mathit{domain}(\lambda x{:}T.\ t) & = & T \\ \mathit{domain}(\texttt{fix } (x{:}T \to T')\ v) & = & T \\ \mathit{domain}(c\ v_1 \ldots v_{i-1}) & = & \text{type of } i^{th} \text{ argument to } c \end{array}$$

The rule [E-Cast-Fn] casts a function v to type $x : S \to T$ by creating a new function:

$$\lambda x{:}S.\ \langle T \rangle\ (v\ (\langle D \rangle\ x))$$

where $D = \mathit{domain}(v)$ is the domain type of the function v. This new function takes a value x of type S, casts it to D, applies the given function v, and casts the result to the desired result type T. Thus, casting a function to a different function type will always succeed, since the domain and range values are checked lazily, in a manner reminiscent of higher-order contracts [13].

For a cast to a refinement type, $\langle \texttt{Refine } T\ f \rangle\ v$, the rule [E-Refine] first casts v to type T via the cast $\langle T \rangle\ v$ and then checks if the predicate f holds on this value. If it does, the cast succeeds and returns $\langle T \rangle\ v$.

Casts to type `*` succeed only for special values of type `*`, via the rule [E-Cast-Type].

The operation `fix` is used to define recursive functions and types, which are considered values, and hence `fix` $U\ v$ is also a value. However, when this construct `fix` $U\ v$ appears in a *strict position* (*i.e.*, in a function position or in a cast), the rule [E-fix] performs one step of unrolling to yield $v\ (\texttt{fix } U\ v)$.

4. Type System

The Sage type system is defined via the type rules and judgments shown in Figure 6. Typing is performed in an environment E that binds variables to types and, in some cases, to values. We assume that variables are bound at most once in an environment and, as usual, we apply implicit α-renaming of bound variables to maintain this assumption and to ensure that substitutions are capture-avoiding.

The Sage type system guarantees *progress* (*i.e.*, that well-typed programs can only get stuck due to failed casts) and *preservation* (*i.e.*, that evaluation of a term preserves its type). The proofs appear in a companion report [22].

The main typing judgment

$$E \vdash t : T$$

[1] A semantic notion of equality for primitive types could provide additional flexibility, although such a notion would clearly be undecidable for higher-order types. In practice, syntactic equality has been sufficient.

Figure 6: Type Rules

$E ::=$ *Environments:*
- $\emptyset$ — empty environment
- $E, x : T$ — environment extension
- $E, x = v : T$ — environment term extension

Type rules $\boxed{E \vdash t \,:\, T}$

$$\frac{}{E \vdash c \,:\, ty(c)} \quad \text{[T-Const]}$$

$$\frac{(x : T) \in E \quad \text{or} \quad (x = v : T) \in E}{E \vdash x \,:\, \{y{:}T \mid\ y = x\}} \quad \text{[T-Var]}$$

$$\frac{E \vdash S \,:\, * \quad E, x : S \vdash t \,:\, T}{E \vdash (\lambda x{:}S.\ t) \,:\, (x{:}S \to T)} \quad \text{[T-Fun]}$$

$$\frac{E \vdash S \,:\, * \quad E, x : S \vdash T \,:\, *}{E \vdash (x{:}S \to T) \,:\, *} \quad \text{[T-Arrow]}$$

$$\frac{E \vdash t_1 \,:\, (x{:}S \to T) \qquad E \vdash t_2 \,:\, S}{E \vdash t_1\ t_2 \,:\, T[x := t_2]} \quad \text{[T-App]}$$

$$\frac{E \vdash v \,:\, S \quad E, (x = v : S) \vdash t \,:\, T}{E \vdash \texttt{let}\ x = v : S\ \texttt{in}\ t \,:\, T[x := v]} \quad \text{[T-Let]}$$

$$\frac{E \vdash t \,:\, S \qquad E \vdash S <: T}{E \vdash t \,:\, T} \quad \text{[T-Sub]}$$

assigns type T to term t in the environment E. In the rule [T-Const], the auxiliary function ty returns the type of the constant c, as defined in Figure 4. The rule [T-Var] for a variable x extracts the type T of x from the environment, and assigns to x the singleton refinement type $\{y{:}T \mid y = x\}$. For a function $\lambda x{:}S.\ t$, the rule [T-Fun] infers the type T of t in an extended environment and returns the dependent function type $x{:}S \to T$, where x may occur free in T. The type $x{:}S \to T$ is itself a term, which is assigned type $*$ by rule [T-Arrow], provided that both S and T have type $*$ in appropriate environments.

The rule [T-App] for an application $(t_1\ t_2)$ first checks that t_1 has a function type $x{:}S \to T$ and that t_2 is in the domain of t_1. The result type is T with all occurrences of the formal parameter x replaced by the actual parameter t_2.

The type rule [T-Let] for $\texttt{let}\ x = v : S\ \texttt{in}\ t$ first checks that the type of the bound value v is S. Then t is typed in an environment that contains both the type and the value of x. These precise bindings are used in the subtype judgment, as described below. Subtyping is allowed at any point in a typing derivation via the rule [T-Sub].

The subtype judgment

$$E \vdash S <: T$$

states that S is a subtype of T in the environment E, and it is defined as the greatest solution to the collection of subtype rules in Figure 7. The rules [S-Refl] and [S-Dyn] allow every type to be a subtype both of itself and of the type `Dynamic`. The rule [S-Fun] for function types checks the usual contravariant/covariant subtype requirements on function domains and codomains. The rule [S-Var] hygienically replaces a variable with the value to which it is bound.

Figure 7: Subtype Rules

Subtype rules $\boxed{E \vdash S <: T}$

$$\frac{}{E \vdash T <: T} \quad \text{[S-Refl]}$$

$$\frac{}{E \vdash T <: \texttt{Dynamic}} \quad \text{[S-Dyn]}$$

$$\frac{E \vdash T_1 <: S_1 \qquad E, x : T_1 \vdash S_2 <: T_2}{E \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad \text{[S-Fun]}$$

$$\frac{E, F[x := v] \vdash S[x := v] <: T[x := v]}{E, x = v : U, F \vdash S <: T} \quad \text{[S-Var]}$$

$$\frac{s \longrightarrow s' \qquad E \vdash \mathcal{C}[s'] <: T}{E \vdash \mathcal{C}[s] <: T} \quad \text{[S-Eval-L]}$$

$$\frac{t \longrightarrow t' \qquad E \vdash S <: \mathcal{C}[t']}{E \vdash S <: \mathcal{C}[t]} \quad \text{[S-Eval-R]}$$

$$\frac{E \vdash S <: T}{E \vdash (\texttt{Refine}\ S\ f) <: T} \quad \text{[S-Ref-L]}$$

$$\frac{E \vdash S <: T \qquad E, x : S \models f\ x}{E \vdash S <: (\texttt{Refine}\ T\ f)} \quad \text{[S-Ref-R]}$$

The remaining rules are less conventional. Rules [S-Eval-L] and [S-Eval-R] state that the subtype relation is closed under evaluation of terms in arbitrary positions. In these rules, $\mathcal{C}$ denotes an arbitrary context:

$$\begin{aligned} \mathcal{C} \quad ::= \quad & \bullet \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \lambda x{:}\mathcal{C}.\ t \mid \lambda x{:}T.\ \mathcal{C} \\ & \mid \texttt{let}\ x = \mathcal{C} : S\ \texttt{in}\ t \mid \texttt{let}\ x = t : \mathcal{C}\ \texttt{in}\ t \\ & \mid \texttt{let}\ x = t : S\ \texttt{in}\ \mathcal{C} \end{aligned}$$

The rule [S-Ref-L] states that, if S is a subtype of T, then any refinement of S is also a subtype of T. When S is a subtype of T, the rule [S-Ref-R] invokes the theorem proving judgment $E \models f\ x$, discussed below, to determine if $f\ x$ is valid for all values x of type S. If so, then S is a subtype of $\texttt{Refine}\ T\ f$.

Our type system is parameterized with respect to the theorem proving judgment

$$E \models t$$

which defines the validity of term t in an environment E. We specify the interface between the type system and the theorem prover via the following axioms (akin to those found in [35]), which are sufficient to prove soundness of the type system. In the following, all environments are assumed to be well-formed [22].

1. Faithfulness: If $t \longrightarrow^* \texttt{true}$ then $E \models t$. If $t \longrightarrow^* \texttt{false}$ then $E \not\models t$.
2. Hypothesis: If $(x : \{y{:}S \mid t\}) \in E$ then $E \models t[y := x]$.
3. Weakening: If $E, G \models t$ then $E, F, G \models t$.
4. Substitution: If $E, (x : S), F \models t$ and $E \vdash s \,:\, S$ then $E, F[x := s] \models t[x := s]$.

5. Exact Substitution: $E, (x = v : S), F \models t$ if and only if $E, F[x := v] \models t[x := v]$.
6. Preservation: If $s \longrightarrow^* t$, then $E \models \mathcal{C}[s]$ if and only if $E \models \mathcal{C}[t]$.
7. Narrowing: If $E, (x : T), F \models t$ and $E \vdash S <: T$ then $E, (x : S), F \models t$.

An alternative to these axioms is to define the validity judgment $E \models t$ directly. In such an approach, we could say that a term t is valid if, for all *closing substitutions* σ that map the names in E to terms consistent with their types, the term $\sigma(t)$ evaluates to true:

$$\frac{\begin{array}{c}\forall \sigma \text{ if } \sigma \text{ is consistent with } E \\ \text{then } \sigma(t) \longrightarrow^* \texttt{true}\end{array}}{E \models t} \quad [\textsc{Validity}]$$

This approach has several drawbacks. First, the rule makes the type system less flexible with regard to the underlying logic. More importantly, however, the rule creates a cyclic dependency between validity and the typing of terms in σ. Thus, consistency of the resulting system is non-obvious and remains an open question. For these reasons, we stick to the original axiomatization of theorem proving.

A consequence of the Faithfulness axiom is that the validity judgment is undecidable. In addition, the subtype judgment may require an unbounded amount of compile-time evaluation. These decidability limitations motivate the development of the hybrid type checking techniques of the following section.

5. Hybrid Type Compilation

The Sage hybrid type checking (or *compilation*) algorithm shown in Figure 8 type checks programs and simultaneously inserts dynamic casts. These casts compensate for inevitable limitations in the Sage subtype algorithm, which is a conservative approximation of the subtype relation.

5.1 Algorithmic Subtyping

For any subtype query $E \vdash S <: T$, the algorithmic subtyping judgment $E \vdash^a_{alg} S <: T$ returns a result $a \in \{\surd, \times, ?\}$ depending on whether the algorithm succeeds in proving ($\surd$) or refuting ($\times$) the subtype query, or whether it cannot decide the query (?). Our algorithm conservatively approximates the subtype specification in Figure 6. However, special care must be taken in the treatment of `Dynamic`. Since we would like values of type `Dynamic` to be implicitly cast to other types, such as `Int`, the subtype algorithm should conclude $E \vdash^?_{alg}$ `Dynamic` $<:$ `Int` (forcing a cast from `Dynamic` to `Int`), even though clearly $E \not\vdash$ `Dynamic` $<:$ `Int`. We thus formalize our requirements for the subtype algorithm as the following lemma.

Lemma 1 (Algorithmic Subtyping).

1. *If* $E \vdash^{\surd}_{alg} S <: T$ *then* $E \vdash S <: T$.
2. *If* $E \vdash^{\times}_{alg} T_1 <: T_2$ *then* $\forall F, S_1, S_2$ *that are obtained from* E, T_1, T_2 *by replacing the type* `Dynamic` *by any type, we have that* $F \not\vdash S_1 <: S_2$.

Clearly, a naïve subtype algorithm could always return the result "?" and thus trivially satisfy these requirements, but more precise results enable Sage to verify more properties and to detect more errors at compile time.

This specification of the subtype algorithm is sufficient for describing the compilation process, and we defer presenting the full details of the algorithm until Section 6.

Figure 8: Compilation Rules

Compilation rules $\boxed{E \vdash s \hookrightarrow t : T}$

$$\frac{(x : T) \in E \quad \text{or} \quad (x = t : T) \in E}{E \vdash x \hookrightarrow x : \{y{:}T \mid y = x\}} \quad [\textsc{C-Var}]$$

$$\frac{}{E \vdash c \hookrightarrow c : ty(c)} \quad [\textsc{C-Const}]$$

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \qquad E, x : S' \vdash t \hookrightarrow t' : T}{E \vdash (\lambda x{:}S.\ t) \hookrightarrow (\lambda x{:}S'.\ t') : (x{:}S' \to T)} \quad [\textsc{C-Fun}]$$

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \qquad E, x : S' \vdash T \hookrightarrow T' \downarrow *}{E \vdash (x{:}S \to T) \hookrightarrow (x{:}S' \to T') : *} \quad [\textsc{C-Arrow}]$$

$$\frac{E \vdash t_1 \hookrightarrow t_1' : U \qquad \mathit{unrefine}(U) = x{:}S \to T \qquad E \vdash t_2 \hookrightarrow t_2' \downarrow S}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : T[x := t_2']} \quad [\textsc{C-App1}]$$

$$\frac{E \vdash t_1 \hookrightarrow t_1' \downarrow (\texttt{Dynamic} \to \texttt{Dynamic}) \qquad E \vdash t_2 \hookrightarrow t_2' \downarrow \texttt{Dynamic}}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : \texttt{Dynamic}} \quad [\textsc{C-App2}]$$

$$\frac{\begin{array}{c}E \vdash S \hookrightarrow S' \downarrow * \qquad E \vdash v \hookrightarrow v' \downarrow S' \\ E, (x = v' : S') \vdash t \hookrightarrow t' : T \qquad T' = T[x := v']\end{array}}{\begin{array}{l}E \vdash \texttt{let}\ x = v : S\ \texttt{in}\ t \\ \quad \hookrightarrow \texttt{let}\ x = v' : S'\ \texttt{in}\ t' : T'\end{array}} \quad [\textsc{C-Let}]$$

Compilation and checking rules $\boxed{E \vdash s \hookrightarrow t \downarrow T}$

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{\surd}_{alg} S <: T}{E \vdash t \hookrightarrow t' \downarrow T} \quad [\textsc{CC-Ok}]$$

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash t \hookrightarrow (\langle T \rangle\ t') \downarrow T} \quad [\textsc{CC-Chk}]$$

Algorithmic subtyping $\boxed{E \vdash^a_{alg} S <: T}$

separate algorithm

5.2 Checking and Compilation

The compilation judgment

$$E \vdash s \hookrightarrow t : T$$

compiles the source term s, in environment E, to a compiled term t (possibly with additional casts), where T is the type of t. The compilation and checking judgment

$$E \vdash s \hookrightarrow t \downarrow T$$

is similar, except that it takes as an input the desired type T and ensures that t has type T.

Many of the compilation rules are similar to the corresponding type rules, *e.g.*, [C-Var] and [C-Const]. The rule [C-Fun] compiles a function $\lambda x{:}S.\ t$ by compiling S to some type S' of type $*$ and then compiling t (in the extended environment $E, x : S'$) to a term t' of type T. The rule returns the compiled function $\lambda x{:}S'.\ t'$ of type $x{:}S' \to T$. The rule [C-Arrow] compiles a function type by compiling the two

component types and checking that they both have type `*`. The rule [C-Let] compiles the term `let` $x = v : S$ `in` t by recursively compiling v, S and t in appropriate environments.

The rules for function application are more interesting. The rule [C-App1] compiles an application $t_1\ t_2$ by compiling the function t_1 to some term t_1' of some type U. The type U may be a function type embedded inside refinements. In order to extract the actual type of the parameter to the function, we use *unrefine* to remove any outer refinements of U before checking the type of the argument t_2 against the expected type. Formally, *unrefine* is defined as follows:

$$\begin{array}{rcll} \mathit{unrefine} : \mathit{Term} & \rightarrow & \mathit{Term} & \\ \mathit{unrefine}(x{:}S \rightarrow T) & = & x{:}S \rightarrow T & \\ \mathit{unrefine}(\texttt{Refine}\ T\ f) & = & \mathit{unrefine}(T) & \\ \mathit{unrefine}(S) & = & \mathit{unrefine}(S') & \text{if } S \longrightarrow S' \end{array}$$

The last clause permits S to be simplified via evaluation while removing outer refinements. Given the expressiveness of the type system, this evaluation may not converge within a given time bound. Hence, to ensure that our compiler accepts all (arbitrarily complicated) well-typed programs, the rule [C-App2] provides a backup compilation strategy for applications that requires less static analysis, but performs more dynamic checking. This rule checks that the function expression has the most general function type `Dynamic` → `Dynamic`, and correspondingly coerces t_2 to type `Dynamic`, resulting in an application with type `Dynamic`.

The rules defining the compilation and checking judgment

$$E \vdash s \hookrightarrow t \downarrow T$$

illustrate the key ideas of hybrid type checking. The rules [CC-Ok] and [CC-Chk] compile the given term and check that the compiled term has the expected type T via the algorithmic subtyping judgment

$$E \vdash^{a}_{alg} S <: T.$$

If this judgment succeeds ($a = \surd$), then [CC-OK] returns the compiled term. If the subtyping judgment is undecided ($a = ?$), then [CC-Chk] encloses the compiled term in the cast $\langle T \rangle$ to preserve dynamic type safety.

The compilation rules guarantee that a compiled program is well-typed [22], and thus compiled programs only go wrong due to failed casts. In addition, this property permits type-directed optimizations on compiled code.

5.3 Example

To illustrate how Sage verifies specifications statically when possible, but dynamically when necessary, we consider the compilation of the following term:

$$t \stackrel{\text{def}}{=} \texttt{printf "\%d" 4}$$

For this term, the rule [C-App1] will first compile the subexpression `(printf "%d")` via the following compilation judgment (based on the type of `printf` from Section 2.4):

$$\emptyset \vdash \texttt{(printf "\%d")} \hookrightarrow \texttt{(printf "\%d") : (Printf\_Args "\%d")}$$

The rule [C-App1] then calls the function *unrefine* to evaluate `(Printf_Args "%d")` to the normal form `Int` → `Unit`. Since 4 has type `Int`, the term t is therefore accepted as is; no casts are needed.

However, the computation for `(Printf_Args "%d")` may not terminate within a preset time limit. In this case, the compiler uses the rule [C-App2] to compile t into the code:

$$(\langle \texttt{Dynamic} \rightarrow \texttt{Dynamic} \rangle\ \texttt{(printf "\%d")})\ \ 4$$

At run time, `(printf "%d")` will evaluate to some function $(\lambda x{:}\texttt{Int}.\ t')$ that expects an `Int`, yielding the application:

$$(\langle \texttt{Dynamic} \rightarrow \texttt{Dynamic} \rangle\ (\lambda x{:}\texttt{Int}.\ t'))\ \ 4$$

The rule [E-Cast-Fn] then reduces this term to:

$$(\lambda x{:}\texttt{Dynamic}.\ \langle \texttt{Dynamic} \rangle\ ((\lambda x{:}\texttt{Int}.\ t')\ (\langle \texttt{Int} \rangle\ x)))\ \ 4$$

where the nested cast $\langle \texttt{Int} \rangle\ x$ dynamically ensures that the next argument to `printf` must be an integer.

6. Implementation

Our prototype Sage implementation consists of roughly 5,000 lines of OCaml code. The run time implements the semantics from Section 3, with one extension for supporting the counter-example database when casts fail. Specifically, suppose the compiler inserts the cast $(\langle T \rangle\ t)$ because it cannot prove or refute some subtype test $E \vdash S <: T$. If that cast fails, the run time inserts an entry into the database asserting that $E \not\vdash S <: T$.

Function casts must be treated with care to ensure blame is assigned appropriately upon failure [13]. In particular, if a cast inserted during the lazy evaluation of a function cast fails, an entry for the original, top-level function cast is inserted into the database rather than for the "smaller" cast on the argument or return value.

The Sage subtype algorithm computes the greatest fixed point of the algorithmic rules in Figure 9. These rules return 3-valued results which are combined with the 3-valued conjunction operator ⊗:

⊗	√	?	×
√	√	?	×
?	?	?	×
×	×	×	×

The algorithm attempts to apply the rules in the order in which they are presented in the figure. If no rule applies, the algorithm returns $E \vdash^{?}_{alg} S <: T$. Most of the rules are straightforward, and we focus primarily on the most interesting rules:

[AS-Db]: Before applying any other rules, the algorithm attempts to refute that $E \vdash S <: T$ by querying the database of previously refuted subtype relationships. The judgment $E \vdash^{\times}_{db} S <: T$ indicates that the database includes an entry stating that S is not a subtype of T in an environment E', where E and E' are compatible in the sense that they include the same bindings for the free variables in S and T. This compatibility requirement ensures that we only re-use a refutation in a typing context in which it is meaningful.

[AS-Eval-L] and [AS-Eval-R]: These two rules evaluate the terms representing types. The algorithm only applies these two rules a bounded number of times before timing out and forcing the algorithm to use a different rule or return "?". This prevents non-terminating computation as well as infinite unrolling of recursive types.

[AS-Dyn-L] and [AS-Dyn-R]: These rules ensure that any type can be considered a subtype of `Dynamic` and that converting from `Dynamic` to any type requires an explicit coercion.

[AS-Ref-R]: This rule for checking whether S is a subtype of a specific refinement type relies on a theorem-proving algorithm, $E \models^{a}_{alg} t$, for checking validity. This algorithm is an approximation of some validity judgment $E \models t$ sat-

Figure 9: Subtyping Algorithm

Algorithmic subtyping rules $\boxed{E \vdash^{a}_{alg} S <: T}$

$$\frac{E \vdash^{\times}_{db} S <: T}{E \vdash^{\times}_{alg} S <: T} \quad [\text{AS-DB}]$$

$$\frac{}{E \vdash^{\surd}_{alg} T <: T} \quad [\text{AS-REFL}]$$

$$\frac{E \vdash^{a}_{alg} T_1 <: S_1 \quad E, x : T_1 \vdash^{b}_{alg} S_2 <: T_2 \quad c = a \otimes b}{E \vdash^{c}_{alg} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad [\text{AS-FUN}]$$

$$\frac{}{E \vdash^{?}_{alg} \texttt{Dynamic} <: T} \quad [\text{AS-DYN-L}]$$

$$\frac{}{E \vdash^{\surd}_{alg} S <: \texttt{Dynamic}} \quad [\text{AS-DYN-R}]$$

$$\frac{E \vdash^{a}_{alg} S <: T \quad a \in \{\surd, ?\}}{E \vdash^{a}_{alg} (\texttt{Refine}\ S\ f) <: T} \quad [\text{AS-REF-L}]$$

$$\frac{E \vdash^{a}_{alg} S <: T \quad E, x : S \models^{b}_{alg} f\ x \quad c = a \otimes b}{E \vdash^{c}_{alg} S <: (\texttt{Refine}\ T\ f)} \quad [\text{AS-REF-R}]$$

$$\frac{E, F[x := v] \vdash^{a}_{alg} S[x := v] <: T[x := v]}{E, x = v : u, F \vdash^{a}_{alg} S <: T} \quad [\text{AS-VAR}]$$

$$\frac{s \longrightarrow s' \quad E \vdash^{a}_{alg} D[s'] <: T}{E \vdash^{a}_{alg} D[s] <: T} \quad [\text{AS-EVAL-L}]$$

$$\frac{t \longrightarrow t' \quad E \vdash^{a}_{alg} S <: D_2[t']}{E \vdash^{a}_{alg} S <: D_2[t]} \quad [\text{AS-EVAL-R}]$$

$D ::= \ \bullet \mid N\ D$ where N is a normal form

Algorithmic theorem proving $\boxed{E \models^{a}_{alg} t}$

separate algorithm

Counter-example database $\boxed{E \vdash^{\times}_{db} S <: T}$

database of previously failed casts

isfying the axioms in Section 4. As with subtyping, the result $a \in \{\surd, ?, \times\}$ indicates whether or not the theorem prover could prove or refute the validity of t. The algorithmic theorem proving judgment must be conservative with respect to the logic it is approximating, as captured in the following requirement:

REQUIREMENT 2 (Algorithmic Theorem Proving).

1. *If* $E \models^{\surd}_{alg} t$ *then* $E \models t$.
2. *If* $E \models^{\times}_{alg} t$ *then* $\forall E', t'$ *obtained from* E *and* t *by replacing the type* `Dynamic` *by any type, we have that* $E' \not\models t'$.

Our current implementation of this theorem-proving algorithm translates the query $E \models^{a}_{alg} t$ into input for the Simplify theorem prover [11]. For example, the query

$$x : \{x{:}\texttt{Int} \mid x \geq 0\} \models^{a}_{alg} x + x \geq 0$$

is translated into the Simplify query:

```
(IMPLIES (>= x 0) (>= (+ x x) 0))
```

for which Simplify returns `Valid`. Given the incompleteness of Simplify (and other theorem provers), care must be taken in how the Simplify results are interpreted. For example, on the translated version of the query

$$x : \texttt{Int} \models^{a}_{alg} x * x \geq 0$$

Simplify returns `Invalid`, because it is incomplete for arbitrary multiplication. In this case, the SAGE theorem prover returns the result "?" to indicate that the validity of the query is unknown. We currently assume that the theorem prover is complete for linear integer arithmetic. Simplify has very effective heuristics for integer arithmetic, but does not fully satisfy this specification; we plan to replace it with an alternative prover that is complete for this domain.

Assuming that $E \models^{a}_{alg} t$ satisfies Requirement 2 and that $E \vdash^{\times}_{db} S <: T$ only if $E \not\vdash S <: T$ (meaning that the database only contains invalid subtype tests), it is straightforward to show that the subtype algorithm $E \vdash^{a}_{alg} S <: T$ satisfies Lemma 1.

7. Experimental Results

We evaluated the SAGE language and implementation using the benchmarks listed in Figure 10. The program `arith.sage` defines and uses a number of mathematical functions, such as `min`, `abs`, and `mod`, where refinement types provide precise specifications. The programs `bst.sage` and `heap.sage` implement and use binary search trees and heaps, and the program `polylist.sage` defines and manipulates polymorphic lists. The types of these data structures ensure that every operation preserves key invariants. The program `stlc.sage` implements a type checker and evaluator for the simply-typed lambda calculus (STLC), where SAGE types specify that evaluating an STLC-term preserves its STLC-type. We also include the sorting algorithm `mergesort.sage`, as well as the `regexp.sage` and `printf.sage` examples discussed earlier.

Figure 10 characterizes the performance of the subtype algorithm on these benchmarks. We consider two configurations of this algorithm, both with and without the theorem prover. For each configuration, the figure shows the number of subtyping judgments proved (denoted by $\surd$), refuted (denoted by $\times$), and left undecided (denoted by ?). The benchmarks are all well-typed, so no subtype queries are refuted. Note that the theorem prover enables SAGE to decide many more subtype queries. In particular, many of the benchmarks include complex refinement types that use integer arithmetic to specify ordering and structure invariants; theorem proving is particularly helpful in verifying these benchmarks.

Our subtyping algorithm performs quite well and verifies a large majority of subtype tests performed by the compiler. Only a small number of undecided queries result in casts. For example, in `regexp.sage`, SAGE cannot statically verify subtyping relations involving regular expressions (they are checked dynamically) but it statically verifies all other sub-

Figure 10: Subtyping Algorithm Statistics

Benchmark	Lines of code	Without Prover √	?	×	With Prover √	?	×
`arith.sage`	45	132	13	0	145	0	0
`bst.sage`	62	344	28	0	372	0	0
`heap.sage`	69	322	34	0	356	0	0
`mergesort.sage`	80	437	31	0	468	0	0
`polylist.sage`	397	2338	5	0	2343	0	0
`printf.sage`	228	321	1	0	321	1	0
`regexp.sage`	113	391	2	0	391	2	0
`stlc.sage`	227	677	11	0	677	11	0
Total	1221	4962	125	0	5073	14	0

type judgments. Some complicated tests in `stlc.sage` and `printf.sage` must also be checked dynamically.

Despite the use of a theorem prover, compilation times for these benchmarks is quite manageable. On a 3GHz Pentium 4 Xeon processor running Linux 2.6.14, compilation required fewer than 10 seconds for each of the benchmarks, except for `polylist.sage` which took approximately 18 seconds. We also measured the number of evaluation steps required during each subtype test. We found that 83% of the subtype tests required no evaluation, 91% required five or fewer steps, and only a handful of the the tests in our benchmarks required more than 50 evaluation steps.

8. Related Work

The enforcement of complex program specifications, or *contracts*, is the subject of a large body of prior work [32, 13, 26, 21, 24, 28, 37, 25, 12, 8]. Since these contracts are typically not expressible in classical type systems, they have previously been relegated to dynamic checking, as in, for example, Eiffel [32]. Eiffel's expressive contract language is strictly separated from its type system. Hybrid type checking extends contracts with the ability to check many properties at compile time. Meunier *et al* have also investigated statically verifying contracts via set-based analysis [31].

The static checking tool ESC/Java [17] supports expressive JML specifications [26]. However, ESC/Java's error messages may be caused either by incorrect programs or by limitations in its own analysis, and thus it may give false alarms on correct (but perhaps complicated) programs. In contrast, hybrid type checking only produces error messages for provably ill-typed programs.

The Spec# programming system extends C# with expressive specifications [6], including preconditions, postconditions, and non-null annotations. Specifications are enforced dynamically, and can be also checked statically via a separate tool. The system is somewhat less tightly integrated than in Sage. For example, successful static verification does not automatically remove the corresponding dynamic checks.

Recent work on advanced type systems has influenced our choice of how to express program invariants. In particular, Freeman and Pfenning [18] extended ML with another form of refinement types. They work focuses on providing both decidable type checking and type inference, instead of on supporting arbitrary refinement predicates.

Xi and Pfenning have explored applications of dependent types in Dependent ML [44, 43]. Decidability of type checking is preserved by appropriately restricting which terms can appear in types. Despite these restrictions, a number of interesting examples can be expressed in Dependent ML. Our system of dependent types extends theirs with arbitrary executable refinement predicates, and the hybrid type checking infrastructure is designed to cope with the resulting undecidability. In a complementary approach, Chen and Xi [10] address decidability limitations by providing a mechanism through which the programmer can provide proofs of subtle properties in the source code.

Recently, Ou, Tan, Mandelbaum, and Walker developed a dependent type system that leverages dynamic checks [35] in a way similar to Sage. Unlike Sage, their system is decidable, and they leverage dynamic checks to reduce the need for precise type annotations in explicitly labeled regions of programs. They consider mutable data, which we intend to add to Sage in the future. We are exploring other language features, such as objects [16], as well.

Barendregt introduced the unification of types and terms, which allows types to be flexibly expressed as complex expressions, while simplifying the underlying theory [5]. The language Cayenne adopts this approach and copes with the resulting undecidability of type checking by allowing a maximum number of steps, somewhat like a timeout, before reporting to the user that typing has failed [4]. Hybrid type checking differs in that instead of rejecting subtly well-typed programs outright, it provisionally accepts them and then performs dynamic checking where necessary.

Other authors have considered pragmatic combinations of both static and dynamic checking. Abadi, Cardelli, Pierce and Plotkin [1] extended a static type system with a type `Dynamic` that could be explicitly cast to and from any other type (with appropriate run-time checks). Henglein characterized the *completion process* of inserting the necessary coercions, and presented a rewriting system for generating minimal completions [23]. Thatte developed a similar system in which the necessary casts are implicit [39]. For Scheme, soft type systems [29, 42, 3, 15] prevent some basic type errors statically, while checking other properties at run time.

The limitations of purely-static and purely-dynamic approaches have also motivated other work on hybrid analyses. For example, CCured [33] is a sophisticated hybrid analysis for preventing the ubiquitous array bounds violations in the C programming language. Unlike our proposed approach, it does not detect errors statically. Instead, the static analysis is used to optimize the run-time analysis. Specialized hybrid analyses have been proposed for other problems as well, such as data race condition checking [41, 34, 2].

9. Conclusions and Future Work

Program specifications are essential for modular development of reliable software. Sage uses a synthesis of first-class types, `Dynamic`, and refinement types to enforce precise specifications in a flexible manner. Our hybrid checking algorithm extends traditional type checking with a theorem prover, a database of counter-examples, and the ability to insert dynamic checks when necessary. Experimental results show that Sage can verify many correctness properties at compile time. We believe that Sage illustrates a promising approach for reliable software development.

A number of opportunities remain for future work. The benefits of the refuted subtype database can clearly be amplified by maintaining a single repository for all local and non-local users of Sage. We also plan to integrate randomized or directed [20] testing to refute additional validity queries, thereby detecting more errors at compile time. Since precise type inference for Sage is undecidable, we plan to develop hybrid algorithms that infer precise types

for most type variables, and that may occasionally infer the looser type `Dynamic` in particularly complicated situations.

Acknowledgments: We thank Robby Findler and Bo Adler for useful feedback on this work.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Symposium on Principles of Programming Languages*, pages 213–227, 1989.

[2] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[3] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[4] L. Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[5] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop*, pages 49–69, 2005.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. 2004.

[8] M. Blume and D. A. McAllester. A sound (and complete) model of contracts. In *International Conference on Functional Programming*, pages 189–200, 2004.

[9] L. Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, DEC Systems Research Center, Palo Alto, California, 1986.

[10] C. Chen and H. Xi. Combining programming with theorem proving. In *International Conference on Functional Programming*, pages 66–77, 2005.

[11] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Research Report HPL-2003-148, HP Labs, 2003.

[12] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Symposium on Logic Programming*, pages 226–241, 2006.

[13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

[14] C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[15] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Finding bugs in the web of program invariants. In *Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

[16] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid object types, specifications, and invariants. In *Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[18] T. Freeman and F. Pfenning. Refinement types for ML. In *Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[19] The Glasgow Haskell Compiler, release 6.4.1. Available from `http://www.haskell.org/ghc`, 2006.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation*, 2005.

[21] B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A language manual for Sather 1.1, 1996.

[22] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications, extended report. Available at `http://www.soe.ucsc.edu/~cormac/papers/sage-full.ps`, 2006.

[23] F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[24] R. C. Holt and J. R. Cordy. The Turing programming language. *Communications of the ACM*, 31:1310–1424, 1988.

[25] M. Kölling and J. Rosenberg. Blue: Language specification, version 0.94, 1997.

[26] G. T. Leavens and Y. Cheon. Design by contract with JML, 2005. avaiable at `http://www.cs.iastate.edu/~leavens/JML/`.

[27] X. Leroy (with D. Doligez, J. Garrigue, D. Rémy and J. Vouillon). The Objective Caml system, release 3.08. `http://caml.inria.fr/pub/docs/manual-ocaml/`, 2004.

[28] D. Luckham. Programming with specifications. *Texts and Monographs in Computer Science*, 1990.

[29] M. Fagan. *Soft Typing*. PhD thesis, Rice University, 1990.

[30] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *International Conference on Functional Programming*, pages 213–225, 2003.

[31] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *Symposium on Principles of Programming Languages*, pages 218–231, 2006.

[32] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[33] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[34] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[35] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

[36] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

[37] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[38] Paul Hudak and Simon Peyton-Jones and Philip Wadler (eds.). Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5), 1992.

[39] S. Thatte. Quasi-static typing. In *Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[40] Sussman, G.J. and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

[41] C. von Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[42] A. Wright and R. Cartwright. A practical soft type system for Scheme. In *Conference on Lisp and Functional Programming*, pages 250–262, 1994.

[43] H. Xi. Imperative programming with dependent types. In *IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

[44] H. Xi and F. Pfenning. Dependent types in practical programming. In *Symposium on Principles of Programming Languages*, pages 214–227, 1999.

From Variadic Functions to Variadic Relations

A miniKanren Perspective

William E. Byrd and Daniel P. Friedman

Department of Computer Science, Indiana University, Bloomington, IN 47408

{webyrd,dfried}@cs.indiana.edu

Abstract

We present an implementation of miniKanren, an embedding of logic programming in R^{5}RS Scheme that comprises three logic operators. We describe these operators, and use them to define *plus*^{o}, a relation that adds two numbers. We then define *plus*^{*o}, which adds zero or more numbers; *plus*^{*o} takes exactly two arguments, the first of which is a list of numbers to be added or a logical variable representing such a list. We call such a relation *pseudo-variadic*. Combining Scheme's var-args facility with pseudo-variadic helper relations leads to *variadic* relations, which take a variable number of arguments. We focus on pseudo-variadic relations, which we demonstrate are more flexible than their variadic equivalents.

We show how to define *plus*^{*o} in terms of *plus*^{o} using *foldr*^{o} and *foldl*^{o}, higher-order relational abstractions derived from Haskell's *foldr* and *foldl* functions. These higher-order abstractions demonstrate the benefit of embedding relational operators in a functional language. We define many other pseudo-variadic relations using *foldr*^{o} and *foldl*^{o}, consider the limitations of these abstractions, and explore their effect on the divergence behavior of the relations they define. We also consider *double-pseudo-variadic* relations, a generalization of pseudo-variadic relations that take as their first argument a list of lists or a logical variable representing a list of lists.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords miniKanren, variadic, pseudo-variadic, double-pseudo-variadic, Scheme, logic programming, relations

1. Introduction

Scheme's var-args mechanism makes it easy to define *variadic functions*, which take a variable number of arguments. miniKanren, an embedding of logic programming in Scheme, makes it easy to define *variadic relations* using that same mechanism. A variadic relation takes a variable number of arguments, but can be defined in terms of a *pseudo-variadic* helper relation that takes only two arguments, the first of which must be a list. A fresh (uninstantiated) logical variable passed as the first argument to a pseudo-variadic relation represents arbitrarily many arguments in the equivalent variadic relation—because of this flexibility, we focus on pseudo-variadic relations instead of their variadic brethren.

Certain variadic functions can be defined in terms of binary functions using the *foldr* or *foldl* abstractions [8]; certain variadic relations can be defined in terms of ternary relations using *foldr*^{o} or *foldl*^{o}, the relational equivalents of *foldr* and *foldl*, respectively. *foldr*^{o}, *foldl*^{o}, and other miniKanren relations can be derived from their corresponding function definitions—we have omitted these derivations, most of which are trivial. The ease with which we can define higher-order relational abstractions such as *foldr*^{o} demonstrates the benefits of using Scheme as the host language for miniKanren.

We also consider *double-pseudo-variadic* relations, which are a generalization of pseudo-variadic relations. A double-pseudo-variadic relation takes two arguments, the first of which is either a list of lists or a logical variable representing a list of lists. (Unless we explicitly state that a variable is a lexical variable, it is assumed to be a logical variable.) As with pseudo-variadic relations, certain double-pseudo-variadic relations can be defined using higher-order relational abstractions.

miniKanren is a descendant of the language presented in *The Reasoned Schemer* [6], which was itself inspired by Prolog. Not surprisingly, the techniques we present will be familiar to most experienced Prolog programmers.

This paper has four additional sections and an appendix. Section 2 gives a brief overview of miniKanren, and presents a simple unary arithmetic system in miniKanren using Peano numerals; these arithmetic relations are used in several of the examples. Readers unfamiliar with logic programming should carefully study this material and the miniKanren implementation in the appendix before reading section 3. (For a gentle introduction to logic programming, we recommend Clocksin [3].) Section 3 is the heart of the paper; it introduces pseudo-variadic relations, and shows how some pseudo-variadic relations can be defined using the *foldr*^{o} or *foldl*^{o} relational abstractions. Section 4 discusses double-pseudo-variadic relations and the *foldr*^{*o} and *foldl*^{*o} relational abstractions. In section 5 we conclude. The appendix presents an R^{5}RS-compliant [9] implementation of miniKanren.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

2. miniKanren Overview

This section is divided into two parts. Part one introduces the three miniKanren operators and demonstrates their behavior. Part two defines several arithmetic relations that are used in later examples, and allows the reader to become familiar with fixed-arity relations before considering the more complicated pseudo-variadic relations.

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. Quotes, quasiquotes, and unquotes are suppressed, and quoted or quasiquoted lists appear with bold parentheses—for example **()** and **(x . x)** are entered as `'()` and `` `(x . ,x) ``, respectively. By our convention, names of relations end with a superscript o—for example $plus^o$, which is entered as `pluso`. miniKanren's relational operators do not follow this convention: ≡ (entered as `==`), **cond**^{e} (entered as `conde`), and **fresh**. Similarly, (**run**^{5} (q) *body*) and (**run*** (q) *body*) are entered as `(run 5 (q) body)` and `(run #f (q) body)`, respectively.

2.1 Introduction to miniKanren

miniKanren, like Schelog [15], is an embedding of logic programming in Scheme. miniKanren extends Scheme with three operators: ≡, **cond**^{e}, and **fresh**. There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

fresh, which syntactically looks like **lambda**, introduces new variables into its scope; ≡ unifies two values. Thus

(**fresh** (x y z) (≡ x z) (≡ 3 y))

would associate x with z and y with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

(**run**^{1} (q) (**fresh** (x y z) (≡ x z) (≡ 3 y))) ⇒ (_.0)

The value returned is a list containing the single value _.0; we say that _.0 is the *reified value* of the fresh variable q. q also remains fresh in

(**run**^{1} (q) (**fresh** (x y) (≡ x q) (≡ 3 y))) ⇒ (_.0)

We can get back other values, of course.

```
(run^1 (y)          (run^1 (q)          (run^1 (y)
  (fresh (x z)        (fresh (x z)        (fresh (x y)
    (≡ x z)             (≡ x z)             (≡ 4 x)
    (≡ 3 y)))           (≡ 3 z)             (≡ x y))
                        (≡ q x)))       (≡ 3 y))
```

Each of these examples returns (3); in the rightmost example, the y introduced by **fresh** is different from the y introduced by **run**. **run** can also return the empty list, indicating that there are no values.

(**run**^{1} (x) (≡ 4 3)) ⇒ ()

We use **cond**^{e} to get several values—syntactically, **cond**^{e} looks like **cond** but without ⇒ or **else**. For example,

```
(run^2 (q)
  (fresh (x y z)
    (conde
      ((≡ (x y z x) q))
      ((≡ (z y x z) q))))) ⇒
((_.0 _.1 _.2 _.0) (_.0 _.1 _.2 _.0))
```

Although the two **cond**^{e}-clauses are different, the values returned are identical. This is because distinct reified fresh variables are assigned distinct numbers, increasing from left to right—the numbering starts over again from zero within each value, which is why the reified value of x is _.0 in the first value but _.2 in the second value.

Here is a simpler example using **cond**^{e}.

```
(run^5 (q)
  (fresh (x y z)
    (conde
      ((≡ a x) (≡ 1 y) (≡ d z))
      ((≡ 2 y) (≡ b x) (≡ e z))
      ((≡ f z) (≡ c x) (≡ 3 y)))
    (≡ (x y z) q))) ⇒
((a 1 d) (b 2 e) (c 3 f))
```

The superscript 5 denotes the maximum length of the resultant list. If the superscript $*$ is used, then there is no maximum imposed. This can easily lead to infinite loops:

```
(run* (q)
  (let loop ()
    (conde
      ((≡ #f q))
      ((≡ #t q))
      ((loop)))))
```

Had the $*$ been replaced by a non-negative integer n, then a list of n alternating #f's and #t's would be returned. The **cond**^{e} succeeds while associating q with #f, which accounts for the first value. When getting the second value, the second **cond**^{e}-clause is tried, and the association made between q and #f is forgotten—we say that q has been *refreshed*. In the third **cond**^{e}-clause, q is refreshed once again.

We now look at several interesting examples that rely on any^o.

```
(define any^o
  (lambda (g)
    (conde
      (g)
      ((any^o g)))))
```

any^o tries g an unbounded number of times.

Here is the first example using any^o.

```
(run* (q)
  (conde
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

This example does not terminate, because the call to any^o succeeds an unbounded number of times. If $*$ is replaced by 5, then instead we get (#t #f #f #f #f). (The user should not be concerned with the order in which values are returned.)

Now consider

```
(run^10 (q)
  (any^o
    (conde
      ((≡ 1 q))
      ((≡ 2 q))
      ((≡ 3 q))))) ⇒
(1 2 3 1 2 3 1 2 3 1)
```

Here the values 1, 2, and 3 are interleaved; our use of any^o ensures that this sequence will be repeated indefinitely.

Here is $always^o$,

```
(define always^o (any^o (≡ #f #f)))
```

along with two **run** expressions that use it.

```
(run^1 (x)                (run^5 (x)
  (≡ #t x)                  (cond^e
  always^o                    ((≡ #t x))
  (≡ #f x))                   ((≡ #f x)))
                            always^o
                            (≡ #f x))
```

The left-hand expression diverges—this is because $always^o$ succeeds an unbounded number of times, and because $(\equiv$ #f $x)$ fails each of those times.

The right-hand expression returns a list of five #f's. This is because both **cond**^{e}-clauses are tried, and both succeed. However, only the second **cond**^{e}-clause contributes to the values returned in this example. Nothing changes if we swap the two **cond**^{e}-clauses. If we change the last expression to $(\equiv$ #t $x)$, we instead get a list of five #t's.

Even if some **cond**^{e}-clauses loop indefinitely, other **cond**^{e}-clauses can contribute to the values returned by a **run** expression. (We are not concerned with Scheme expressions looping indefinitely, however.) For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns **(**1 2 3**)**; replacing **run**^{3} with **run**^{4} causes divergence, however, since there are only three values, and since $never^o$ loops indefinitely.

2.2 Peano Arithmetic

The arithmetic examples in this paper use *Peano representation* of numbers (technically, *Peano numerals*). The advantage of this representation is that we can use ≡ both to construct and to match against numbers.

The Peano representation of zero is z, while the immediate successor to a Peano number n is represented as **(**s n**)**. For example, one is the immediate successor of zero—the Peano representation of one is therefore **(**s z**)**. Two is the immediate successor of one, so the Peano representation of two is **(**s **(**s z**))**.

Typographically, we indicate a Peano number using corner brackets—for example, ⌜3⌝ for **(**s **(**s **(**s z**)))**. We represent **(**s x**)** as ⌜x+1⌝, **(**s **(**s x**))** as ⌜x+2⌝, and so forth, where x is a variable or a reified variable (that is, a symbol).

Here is $plus^o$, which adds two Peano numbers.

```
(define plus^o
  (lambda (n m sum)
    (cond^e
      ((≡ ⌜0⌝ n) (≡ m sum))
      ((fresh (x y)
         (≡ ⌜x+1⌝ n)
         (≡ ⌜y+1⌝ sum)
         (plus^o x m y))))))
```

$plus^o$ allows us to find all pairs of numbers that sum to six.

```
(run* (q)
  (fresh (n m)
    (plus^o n m ⌜6⌝)
    (≡ (n m) q))) ⇒
((⌜0⌝ ⌜6⌝)
 (⌜1⌝ ⌜5⌝)
 (⌜2⌝ ⌜4⌝)
 (⌜3⌝ ⌜3⌝)
 (⌜4⌝ ⌜2⌝)
 (⌜5⌝ ⌜1⌝)
 (⌜6⌝ ⌜0⌝))
```

Let us define $minus^o$ using $plus^o$, and use it to find ten pairs of numbers whose difference is six.

```
(define minus^o
  (lambda (n m k)
    (plus^o m k n)))

(run^10 (q)
  (fresh (n m)
    (minus^o n m ⌜6⌝)
    (≡ (n m) q))) ⇒
((⌜6⌝ ⌜0⌝)
 (⌜7⌝ ⌜1⌝)
 (⌜8⌝ ⌜2⌝)
 (⌜9⌝ ⌜3⌝)
 (⌜10⌝ ⌜4⌝)
 (⌜11⌝ ⌜5⌝)
 (⌜12⌝ ⌜6⌝)
 (⌜13⌝ ⌜7⌝)
 (⌜14⌝ ⌜8⌝)
 (⌜15⌝ ⌜9⌝))
```

We have chosen to have subtraction of a larger number from a smaller number fail, rather than be zero.

```
(run* (q) (minus^o ⌜5⌝ ⌜6⌝ q)) ⇒ ()
```

We will also need $even^o$ and $positive^o$ in several examples below.

```
(define even^o
  (lambda (n)
    (cond^e
      ((≡ ⌜0⌝ n))
      ((fresh (m)
         (≡ ⌜m+2⌝ n)
         (even^o m))))))

(define positive^o
  (lambda (n)
    (fresh (m)
      (≡ ⌜m+1⌝ n))))
```

$even^o$ and $positive^o$ ensure that their arguments represent even and positive Peano numbers, respectively.

```
(run^4 (q) (even^o q)) ⇒ (⌜0⌝ ⌜2⌝ ⌜4⌝ ⌜6⌝)

(run* (q) (positive^o q)) ⇒ (⌜-0+1⌝)
```

The value $\ulcorner_{-0}+1\urcorner$ shows that $n + 1$ is positive for every number n.

3. Pseudo-Variadic Relations

Just as a Scheme function can be variadic, so can a miniKanren relation. For example, it is possible to define a variadic version of $plus^o$ using Scheme's var-args feature. We must distinguish between the arguments whose values are to be added and the single argument representing the sum of those values. The simplest solution is to make the first argument represent the sum.

```
(define variadic-plus*o
  (lambda (out . in*)
    (plus*o in* out)))

(define plus*o
  (lambda (in* out)
    (conde
      ((≡ () in*) (≡ ⌜0⌝ out))
      ((fresh (a d res)
         (≡ (a . d) in*)
         (plus^o a res out)
         (plus*o d res))))))
```

Here we use *variadic-plus*^{*o} to find the sum of three, four, and two:

```
(run* (q) (variadic-plus*o q ⌜3⌝ ⌜4⌝ ⌜2⌝)) ⇒ (⌜9⌝)
```

Let us find the number that, when added to four, one, and two, produces nine.

```
(run* (q) (variadic-plus*o ⌜9⌝ ⌜4⌝ q ⌜1⌝ ⌜2⌝)) ⇒ (⌜2⌝)
```

variadic-plus^{*o} is not as general as it could be, however. We cannot, for example, use *variadic-plus*^{*o} to find all sequences of numbers that sum to five. This is because in^* must be an actual list, and cannot be a variable representing a list. The solution is simple—just use *plus*^{*o} in place of *variadic-plus*^{*o}.

plus^{*o}, which does not use Scheme's var-args functionality, takes exactly two arguments. The first argument must be a list of numbers, or a variable representing a list of numbers. The second argument represents the sum of the numbers in the first argument, and can be either a number or a variable representing a number.

Variadic relations, such as *variadic-plus*^{*o}, are defined using *pseudo-variadic* helper relations, such as *plus*^{*o}. Henceforth, we focus exclusively on the more flexible pseudo-variadic relations, keeping in mind that each pseudo-variadic relation can be paired with a variadic relation.

To add three, four, and two using *plus*^{*o}, we write

```
(run* (q) (plus*o (⌜3⌝ ⌜4⌝ ⌜2⌝) q)) ⇒ (⌜9⌝)
```

Here is another way to add three, four, and two.

```
(run¹ (q)
  (fresh (x y z)
    (plus*o x q)
    (≡ (⌜3⌝ . y) x)
    (≡ (⌜4⌝ . z) y)
    (≡ (⌜2⌝ . ()) z))) ⇒ (⌜9⌝)
```

Instead of passing a fully instantiated list of numbers as the first argument to *plus*^{*o}, we pass the fresh variable x—only afterwards do we instantiate x. Each call to ≡ further constrains the possible values of x, and consequently constrains the possible values of q. This technique of constraining the first argument to a pseudo-variadic relation through repeated calls to ≡ is similar to the partial application of a curried function—the *plus*^{*o} relation can be considered both curried and (pseudo) variadic.

Replacing $\mathbf{run}^1$ with $\mathbf{run}^2$ causes the expression to diverge. This is because there is no second value to be found. Although (*plus*^{*o} x q) succeeds an unbounded number of times, after each success one of the calls to ≡ fails, resulting in infinitely many failures without a single success, and therefore divergence. If the call to *plus*^{*o} is made after the calls to ≡, the expression terminates even when using $\mathbf{run}^*$.

```
(run* (q)
  (fresh (x y z)
    (≡ (⌜2⌝ . ()) z)
    (≡ (⌜3⌝ . y) x)
    (≡ (⌜4⌝ . z) y)
    (plus*o x q))) ⇒ (⌜9⌝)
```

We have also reordered the calls to ≡ to illustrate that the list associated with x need not be extended in left-to-right order—this reordering does not affect the behavior of the expression. The three calls to ≡ fully instantiate x; instead, we could have associated z with a pair whose *cdr* is a fresh variable, thereby leaving the variable x only partially instantiated. By the end of section 3, the reader should be able to predict the effect of this change on the behavior of the $\mathbf{run}^*$ expression.

Here is a simpler example—we prove there is no sequence of numbers that begins with five whose sum is three.

```
(run* (q) (plus*o (⌜5⌝ . q) ⌜3⌝)) ⇒ ()
```

Returning to the problem that led us to use *plus*^{*o}, we generate lists of numbers whose sum is five.

```
(run10 (q) (plus*o q ⌜5⌝)) ⇒
((⌜5⌝)
 (⌜5⌝ ⌜0⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜5⌝)
 (⌜1⌝ ⌜4⌝)
 (⌜2⌝ ⌜3⌝)
 (⌜3⌝ ⌜2⌝)
 (⌜4⌝ ⌜1⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜5⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝))
```

There are infinitely many values, since a list can contain arbitrarily many zeros. We will consider the problem of generating all lists of positive numbers whose sum is five, but first we introduce a convenient abstraction for defining pseudo-variadic relations.

3.1 The *foldr*^{o} Abstraction

We can define certain pseudo-variadic relations using the *foldr*^{o} relational abstraction. *foldr*^{o} is derived from *foldr*, a standard abstraction for defining variadic functions in terms of binary ones [8].

```
(define foldr
  (lambda (f)
    (lambda (base-value)
      (letrec ((foldr (lambda (in*)
                        (cond
                          ((null? in*) base-value)
                          (else (f (car in*)
                                   (foldr (cdr in*)))))))))
        foldr))))
```

Here we use *foldr*^{o} to define *plusr*^{*o}, which behaves like *plus*^{*o}. (For another approach to higher order relations such as *foldr*^{o}, see Naish [13] and O'Keefe [14].)

```
(define foldr^o
  (lambda (rel^o)
    (lambda (base-value)
      (letrec ((foldr^o (lambda (in^* out)
                          (cond^e
                            ((≡ () in^*) (≡ base-value out))
                            ((fresh (a d res)
                               (≡ (a . d) in^*)
                               (rel^o a res out)
                               (foldr^o d res)))))))
        foldr^o))))

(define plusr^{*o} ((foldr^o plus^o) ⌜0⌝))
```

The first argument to *foldr* must be a binary function, whereas the first argument to *foldr*^{o} must be a ternary relation. The values of *rel*^{o} and *base-value* do not change in the recursive call to *foldr*^{o}—this allows us to pass in *rel*^{o} and *base-value* before passing in *in** and *out*. We make a distinction between the *rel*^{o} and *base-value* arguments: although *base-value* might be a variable, the value of *rel*^{o} must be a miniKanren relation, and therefore a Scheme function.

We use *positive-plusr*^{*o} to ensure that we add only positive numbers.

```
(define positive-plusr^{*o}
  ((foldr^o (lambda (a res out)
              (fresh ()
                (positive^o a)
                (plus^o a res out))))
   ⌜0⌝))
```

Finally, we have the sixteen lists of positive numbers whose sum is five.

```
(run^* (q) (positive-plusr^{*o} q ⌜5⌝)) ⇒
((⌜5⌝)
 (⌜1⌝ ⌜4⌝)
 (⌜2⌝ ⌜3⌝)
 (⌜1⌝ ⌜1⌝ ⌜3⌝)
 (⌜3⌝ ⌜2⌝)
 (⌜1⌝ ⌜2⌝ ⌜2⌝)
 (⌜4⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜3⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝))
```

Let us consider another pseudo-variadic relation; *positive-even-plusr*^{*o} succeeds if its first argument represents a list of positive numbers whose sum is even.

```
(define positive-even-plusr^{*o}
  (lambda (in^* out)
    (fresh ()
      (even^o out)
      (positive-plusr^{*o} in^* out))))
```

Here are the first ten values returned by *positive-even-plusr*^{*o}.

```
(run^10 (q)
  (fresh (x y)
    (≡ (x y) q)
    (positive-even-plusr^{*o} x y))) ⇒
((() ⌜0⌝)
 ((⌜2⌝) ⌜2⌝)
 ((⌜1⌝ ⌜1⌝) ⌜2⌝)
 ((⌜4⌝) ⌜4⌝)
 ((⌜1⌝ ⌜3⌝) ⌜4⌝)
 ((⌜2⌝ ⌜2⌝) ⌜4⌝)
 ((⌜3⌝ ⌜1⌝) ⌜4⌝)
 ((⌜1⌝ ⌜1⌝ ⌜2⌝) ⌜4⌝)
 ((⌜1⌝ ⌜2⌝ ⌜1⌝) ⌜4⌝)
 ((⌜2⌝ ⌜1⌝ ⌜1⌝) ⌜4⌝))
```

Replacing **run**^{10} with **run*** causes divergence, since there are infinitely many values.

Let us consider another pseudo-variadic relation defined using *foldr*^{o}. Here is *append*^{o}, which appends two lists, and its pseudo-variadic variant *appendr*^{*o}.

```
(define append^o
  (lambda (l s out)
    (cond^e
      ((≡ () l) (≡ s out))
      ((fresh (a d res)
         (≡ (a . d) l)
         (≡ (a . res) out)
         (append^o d s res))))))

(define appendr^{*o} ((foldr^o append^o) ()))
```

Here are four examples of *appendr*^{*o}. In the first example, we use *appendr*^{*o} simply to append two lists.

```
(run^* (q) (appendr^{*o} ((a b c) (d e)) q)) ⇒ ((a b c d e))
```

In the second example we infer for which value of q the list (a b c d . q) is equal to the concatenation of the lists (a b c), (d e), and (f g).

```
(run^* (q) (appendr^{*o} ((a b c) (d e) (f g)) (a b c d . q)))
⇒ ((e f g))
```

The third example is more interesting—the contents of the second of three lists being appended can be inferred from the second argument to *appendr*^{*o}.

```
(run^* (q) (appendr^{*o} ((a b c) q (g h)) (a b c d e f g h)))
⇒ ((d e f))
```

The final example shows a few of the lists of lists whose contents, when appended, are (2 3 d e).

```
(run^10 (q) (appendr^{*o} ((a b) (c) (1) . q) (a b c 1 2 3 d e)))
⇒
(((2 3 d e))
 ((2 3 d e) ())
 ((2 3 d e) () ())
 (() (2 3 d e))
 ((2) (3 d e))
 ((2 3) (d e))
 ((2 3 d) (e))
 ((2 3 d e) () () ())
 ((2 3 d e) () () () ())
 (() (2 3 d e) ()))
```

Replacing **run**$^{\mathbf{10}}$ with **run*** causes the last expression to diverge, since there are infinitely many values that contain the empty list—by using *pair-appendr*^{*o} instead of *appendr*^{*o}, we filter out these values.

```
(define pair^o
  (lambda (p)
    (fresh (a d)
      (≡ (a . d) p))))

(define pair-appendr^*o
  ((foldr^o (lambda (a res out)
              (fresh ()
                (pair^o a)
                (append^o a res out))))
   ()))
```

Now let us re-evaluate the previous example.

```
(run* (q)
  (pair-appendr^*o ((a b) (c) (1) . q) (a b c 1 2 3 d e))) ⇒
(((2 3 d e))
 ((2) (3 d e))
 ((2 3) (d e))
 ((2 3 d) (e))
 ((2) (3) (d e))
 ((2) (3 d) (e))
 ((2 3) (d) (e))
 ((2) (3) (d) (e)))
```

These eight values are the only ones that do not include the empty list.

3.2 The *foldl*^{o} Abstraction

Previously we defined pseudo-variadic relations with the *foldr*^{o} relational abstraction. We can also define certain pseudo-variadic relations using the *foldl*^{o} relational abstraction, which is derived from the standard *foldl* function; like *foldr*, *foldl* is used to define variadic functions in terms of binary ones.

```
(define foldl
  (lambda (f)
    (letrec
      ((foldl
        (lambda (acc)
          (lambda (in*)
            (cond
              ((null? in*) acc)
              (else ((foldl (f acc (car in*)))
                     (cdr in*)))))))))
      foldl)))
```

Here we use *foldl*^{o} to define *plusl*^{*o} and *appendl*^{*o}, which are similar to *plusr*^{*o} and *appendr*^{*o}, respectively.

```
(define foldl^o
  (lambda (rel^o)
    (letrec
      ((foldl^o (lambda (acc)
                  (lambda (in* out)
                    (cond^e
                      ((≡ () in*) (≡ acc out))
                      ((fresh (a d res)
                         (≡ (a . d) in*)
                         (rel^o acc a res)
                         ((foldl^o res) d out))))))))
      foldl^o)))

(define plusl^*o ((foldl^o plus^o) ⌜0⌝))

(define appendl^*o ((foldl^o append^o) ()))
```

As we did with *foldr*^{o}, we separate the *rel*^{o} and *acc* arguments from the *in** and *out* arguments.

We have defined pseudo-variadic versions of *plus*^{o} using both *foldr*^{o} and *foldl*^{o}; these definitions differ in their divergence behavior. Consider this example, which uses *plusr*^{*o}.

```
(run^1 (q) (plusr^*o (⌜4⌝ q ⌜3⌝) ⌜5⌝)) ⇒ ()
```

Replacing *plusr*^{*o} with *plusl*^{*o} causes the expression to diverge. *foldl*^{o} passes the fresh variable *res* as the third argument to the ternary relation, while *foldr*^{o} instead passes the *out* variable, which in this example is fully instantiated. This accounts for the difference in divergence behavior—the relation called by *foldr*^{o} has additional information that can lead to termination.

It is possible to use *foldl*^{o} to define *positive-plusl*^{*o}, *positive-even-plusl*^{*o}, and *pair-appendl*^{*o}. Some pseudo-variadic relations can be defined using *foldl*^{o}, but not *foldr*^{o}. For example, here is *subsetl*^{o}, which generates subsets of a given set (where sets are represented as lists).

```
(define ess^o
  (lambda (in* x out)
    (cond^e
      ((≡ () in*) (≡ ((x)) out))
      ((fresh (a d â d̂)
         (≡ (a . d) in*)
         (cond^e
           ((≡ (â . d) out) (≡ (x . a) â))
           ((≡ (a . d̂) out) (ess^o d x d̂))))))))

(define subsetl^o ((foldl^o ess^o) ()))
```

Here we use *subsetl*^{o} to find all the subsets of the set containing a, b, and c.

```
(run* (q) (subsetl^o (a b c) q)) ⇒
(((c b a)) ((b a) (c)) ((c a) (b)) ((a) (c b)) ((a) (b) (c)))
```

It is possible to infer the original set from which a given subset has been generated.

```
(run^1 (q) (subsetl^o q ((a d) (c) (b)))) ⇒ ((d c b a))
```

Replacing **run**$^{\mathbf{1}}$ with **run**$^{\mathbf{2}}$ yields two values: (d c b a) and (d c a b). Unfortunately, these values are duplicates—they represent the same set. It is possible to eliminate these duplicate sets (for example, by using Prolog III-style disequality constraints [4]), but the techniques involved are not directly related to pseudo-variadic relations.

Here is *partition-suml*^{o}, whose definition is very similar to that of *subsetl*^{o}. Like *subsetl*^{o}, *partition-suml*^{o} cannot be defined using *foldr*^{o}.

```
(define pes^o
  (lambda (in* x out)
    (cond^e
      ((≡ () in*) (≡ (x) out))
      ((fresh (a d â d̂)
         (≡ (a . d) in*)
         (cond^e
           ((≡ (â . d) out) (plus^o x a â))
           ((≡ (a . d̂) out) (pes^o d x d̂))))))))

(define partition-suml^o ((foldl^o pes^o) ()))
```

partition-suml^o partitions a set of numbers, and returns another set containing the sums of the numbers in the various partitions. (This problem was posed in a July 5, 2006 post on the `comp.lang.scheme` newsgroup [5]). An example helps clarify the problem.

(**run*** (q) (*partition-suml^o* (⌜1⌝ ⌜2⌝ ⌜5⌝ ⌜9⌝) q)) ⇒

((⌜8⌝ ⌜9⌝)
 (⌜3⌝ ⌜5⌝ ⌜9⌝)
 (⌜17⌝)
 (⌜12⌝ ⌜5⌝)
 (⌜3⌝ ⌜14⌝)
 (⌜10⌝ ⌜2⌝ ⌜5⌝)
 (⌜1⌝ ⌜2⌝ ⌜5⌝ ⌜9⌝)
 (⌜6⌝ ⌜2⌝ ⌜9⌝)
 (⌜1⌝ ⌜11⌝ ⌜5⌝)
 (⌜1⌝ ⌜7⌝ ⌜9⌝)
 (⌜1⌝ ⌜2⌝ ⌜14⌝)
 (⌜15⌝ ⌜2⌝)
 (⌜6⌝ ⌜11⌝)
 (⌜10⌝ ⌜7⌝)
 (⌜1⌝ ⌜16⌝))

Consider the value (⌜15⌝ ⌜2⌝). We obtain the ⌜15⌝ by adding ⌜1⌝, ⌜5⌝, and ⌜9⌝, while the ⌜2⌝ is left unchanged.

We can infer the original set of numbers, given a specific final value.

(**run**^{10} (q) (*partition-suml^o* q (⌜3⌝))) ⇒

((⌜3⌝)
 (⌜3⌝ ⌜0⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜0⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜2⌝ ⌜1⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝)
 (⌜1⌝ ⌜0⌝ ⌜2⌝))

There are infinitely many values containing zero—one way of eliminating these values is to use *positive-partition-suml^o*.

```
(define positive-pes^o
  (lambda (in* x out)
    (fresh ()
      (positive^o x)
      (cond^e
        ((≡ () in*) (≡ (x) out))
        ((fresh (a d â d̂)
           (≡ (a . d) in*)
           (cond^e
             ((≡ (â . d) out) (plus^o x a â))
             ((≡ (a . d̂) out) (positive-pes^o d x d̂)))))))))

(define positive-partition-suml^o ((foldl^o positive-pes^o) ()))

(run^4 (q) (positive-partition-suml^o q (⌜3⌝))) ⇒
((⌜3⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

We have eliminated values containing zeros, but we still are left with duplicate values—worse, the last value is not even a set. As before, we could use disequality constraints or other techniques to remove these undesired values. Regardless of whether we use these techniques, the previous **run** expression will diverge if we replace **run**^{4} with **run**^{5}; this divergence is due to our use of *foldl^o*.

3.3 When *foldr^o* and *foldl^o* do not work

Consider *minusr^{*o}*, which is a pseudo-variadic *minus^o* defined using *plusr^{*o}*. Unlike the pseudo-variadic addition relations, *minusr^{*o}* fails when *in** is the empty list. Also, when *in** contains only a single number, that number must be zero. This is because the negation of any positive number is negative, and because Peano numbers only represent non-negative integers. These special cases prevent us from defining *minusr^{*o}* using *foldr^o* or *foldl^o*.

```
(define minusr^*o
  (lambda (in* out)
    (cond^e
      ((≡ (⌜0⌝) in*) (≡ ⌜0⌝ out))
      ((fresh (a d res)
         (≡ (a . d) in*)
         (pair^o d)
         (minus^o a res out)
         (plusr^*o d res))))))
```

Here we use *minusr^{*o}* to generate lists of numbers that, when subtracted from seven, yield three.

(**run**^{14} (q) (*minusr^{*o}* (⌜7⌝ . q) ⌜3⌝)) ⇒

((⌜4⌝)
 (⌜4⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜4⌝)
 (⌜1⌝ ⌜3⌝)
 (⌜2⌝ ⌜2⌝)
 (⌜3⌝ ⌜1⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜4⌝ ⌜0⌝)
 (⌜1⌝ ⌜3⌝ ⌜0⌝)
 (⌜2⌝ ⌜2⌝ ⌜0⌝)
 (⌜3⌝ ⌜1⌝ ⌜0⌝)
 (⌜4⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝))

The values containing zero are not very interesting—let us filter out those values by using *positive-minusr^{*o}*.

```
(define positive-minusr^*o
  (lambda (in* out)
    (fresh (a d res)
      (≡ (a . d) in*)
      (positive^o a)
      (minus^o a res out)
      (positive-plusr^*o d res))))
```

Now we can use **run*** instead of **run**^{14}, since there are only finitely many values.

(**run*** (q) (*positive-minusr^{*o}* (⌜7⌝ . q) ⌜3⌝)) ⇒

((⌜4⌝)
 (⌜1⌝ ⌜3⌝)
 (⌜2⌝ ⌜2⌝)
 (⌜3⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝))

As might be expected, we could use $plusl^{*o}$ to define the relations $minusl^{*o}$ and $positive\text{-}minusl^{*o}$.

Here is $positive^{*o}$, another pseudo-variadic relation that cannot be defined using $foldr^{o}$ or $foldl^{o}$; this is because $positive^{*o}$ takes only one argument.

```
(define positive*o
  (lambda (in*)
    (conde
      ((≡ () in*))
      ((fresh (a d)
         (≡ (a . d) in*)
         (positive^o a)
         (positive*o d))))))

(run^5 (q) (positive*o q)) ⇒
(()
 (⌜-0+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝ ⌜-2+1⌝)
 (⌜-0+1⌝ ⌜-1+1⌝ ⌜-2+1⌝ ⌜-3+1⌝))
```

4. Double-Pseudo-Variadic Relations

A pseudo-variadic relation takes a list, or a variable representing a list, as its first argument; a *double-pseudo-variadic* relation takes a list of lists, or a variable representing a list of lists, as its first argument. Let us define $plusr^{**o}$, the double-pseudo-variadic version of $plusr^{*o}$. We define $plusr^{**o}$ using the $foldr^{*o}$ relational abstraction.

```
(define foldr*o
  (lambda (rel^o)
    (lambda (base-value)
      (letrec ((foldr*o (lambda (in** out)
                          (conde
                            ((≡ () in**) (≡ base-value out))
                            ((fresh (dd)
                               (≡ (() . dd) in**)
                               (foldr*o dd out)))
                            ((fresh (a d dd res)
                               (≡ ((a . d) . dd) in**)
                               (rel^o a res out)
                               (foldr*o (d . dd) res)))))))
        foldr*o))))

(define plusr**o ((foldr*o plus^o) ⌜0⌝))
```

As with $plusr^{*o}$, we can use $plusr^{**o}$ to add three, four, and two.

```
(run* (q) (plusr**o ((⌜3⌝ ⌜4⌝ ⌜2⌝)) q)) ⇒ (⌜9⌝)
```

$plusr^{**o}$ allows us to add three, four, and two in more than one way, by partitioning the list of numbers to be added into various sublists, which can include the empty list.

```
(run* (q) (plusr**o (() (⌜3⌝ ⌜4⌝) () (⌜2⌝)) q)) ⇒ (⌜9⌝)
```

In the previous section we used $plusr^{*o}$ to generate lists of numbers whose sum is five; here we use $plusr^{**o}$ to generate lists of numbers whose sum is three.

```
(run^10 (q) (plusr**o (q) ⌜3⌝)) ⇒
((⌜3⌝)
 (⌜3⌝ ⌜0⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜3⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝ ⌜0⌝)
 (⌜0⌝ ⌜3⌝ ⌜0⌝)
 (⌜1⌝ ⌜2⌝ ⌜0⌝))
```

There are infinitely many such lists, since each list can contain an arbitrary number of zeros.

As we did with $plusr^{*o}$, let us use $plusr^{**o}$ to prove that there is no sequence of numbers that begins with five whose sum is three.

```
(run^1 (q) (plusr**o ((⌜5⌝) q) ⌜3⌝)) ⇒ ()
```

This expression terminates because $plusr^{**o}$ calls ($plus^{o}$ ⌜5⌝ *res* ⌜3⌝), which immediately fails.

Swapping the positions of the fresh variable q and the list containing five yields the expression

```
(run^1 (q) (plusr**o (q (⌜5⌝)) ⌜3⌝))
```

which diverges. q represents a list of numbers—since each list can contain arbitrarily many zeros, there are infinitely many such lists whose sum is less than or equal to three. For each such list, $plusr^{**o}$ sums the numbers in the list, and then adds five to that sum; this fails, of course, since the new sum is greater than three. Since there are infinitely many lists, and therefore infinitely many failures without a single success, the expression diverges.

If we were to restrict q to a list of positive numbers, the previous expression would terminate.

```
(define positive-plusr**o
  ((foldr*o (lambda (a res out)
              (fresh ()
                (positive^o a)
                (plus^o a res out))))
   ⌜0⌝))

(run^1 (q) (positive-plusr**o (q (⌜5⌝)) ⌜3⌝)) ⇒ ()
```

We can now generate all the lists of positive numbers whose sum is three.

```
(run* (q) (positive-plusr**o (q) ⌜3⌝)) ⇒
((⌜3⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

The following expression returns all lists of positive numbers containing five that sum to eight.

```
(run* (q)
  (fresh (x y)
    (positive-plusr**o (x (⌜5⌝) y) ⌜8⌝)
    (appendr*o (x (⌜5⌝) y) q))) ⇒
((⌜5⌝ ⌜3⌝)
 (⌜5⌝ ⌜2⌝ ⌜1⌝)
 (⌜5⌝ ⌜1⌝ ⌜2⌝)
 (⌜5⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜1⌝ ⌜5⌝ ⌜2⌝)
 (⌜1⌝ ⌜5⌝ ⌜1⌝ ⌜1⌝)
 (⌜2⌝ ⌜5⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜5⌝ ⌜1⌝)
 (⌜3⌝ ⌜5⌝)
 (⌜1⌝ ⌜2⌝ ⌜5⌝)
 (⌜2⌝ ⌜1⌝ ⌜5⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜5⌝))
```

Here is a more complicated example—we want to find all lists of numbers that sum to twenty-five and satisfy certain additional constraints. The list must begin with a list w of positive numbers, followed by the number three, followed by any single positive number x, the number four, a list y of positive numbers, the number five, and a list z of positive numbers.

```
(run* (q)
  (fresh (w x y z in**)
    (≡ (w (⌜3⌝ x ⌜4⌝) y (⌜5⌝) z) in**)
    (positive-plusr**o in** ⌜25⌝)
    (appendr*o in** q)))
```

Here is a list of the first four values.

```
((⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜12⌝)
 (⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜1⌝ ⌜11⌝)
 (⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜2⌝ ⌜10⌝)
 (⌜3⌝ ⌜2⌝ ⌜4⌝ ⌜5⌝ ⌜11⌝))
```

For the curious, the 7,806th value is

```
(⌜1⌝ ⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜5⌝ ⌜11⌝)
```

and the 4,844th value is

```
(⌜3⌝ ⌜1⌝ ⌜4⌝ ⌜1⌝ ⌜5⌝ ⌜9⌝ ⌜2⌝)
```

foldr^{*o} is not the only double-pseudo-variadic relational abstraction; here is *foldl*^{*o}, which we use to define *plusl*^{**o} and *positive-plusl*^{**o}.

```
(define foldl*o
  (lambda (rel^o)
    (letrec
      ((foldl*o (lambda (acc)
                  (lambda (in** out)
                    (cond^e
                      ((≡ () in**) (≡ acc out))
                      ((fresh (dd)
                         (≡ (() . dd) in**)
                         ((foldl*o acc) dd out)))
                      ((fresh (a d dd res)
                         (≡ ((a . d) . dd) in**)
                         (rel^o acc a res)
                         ((foldl*o res) (d . dd) out)))))))))
       foldl*o)))

(define plusl**o ((foldl*o plus^o) ⌜0⌝))

(define positive-plusl**o
  ((foldl*o (lambda (acc a res)
              (fresh ()
                (positive^o a)
                (plus^o acc a res))))
   ⌜0⌝))
```

Let us revisit an example demonstrating *positive-plusr*^{**o}; we replace *positive-plusr*^{**o} with *positive-plusl*^{**o}, and **run*** with **run**^{4}.

```
(run^4 (q) (positive-plusl**o (q) ⌜3⌝)) ⇒
((⌜3⌝)
 (⌜1⌝ ⌜2⌝)
 (⌜2⌝ ⌜1⌝)
 (⌜1⌝ ⌜1⌝ ⌜1⌝))
```

We get back the same values as before, although in a different order. If we replace **run**^{4} with **run**^{5}, however, the expression diverges. This should not be surprising, since we have already seen a similar difference in divergence behavior between *plusr*^{*o} and *plusl*^{*o}.

Finally, let us consider a double-pseudo-variadic relation that cannot be defined using *foldr*^{*o} or *foldl*^{*o}. Here is *minusr*^{**o}, the generalization of *minusr*^{*o}.

```
(define minusr**o
  (lambda (in** out)
    (fresh (in*)
      (appendr*o in** in*)
      (minusr*o in* out))))
```

The definition is made simple by the call to *appendr*^{*o}—why do we not use this technique when defining other double-pseudo-variadic relations? Because the call to *appendr*^{*o} can succeed an unbounded number of times when *in*** is fresh or partially instantiated, which can easily lead to divergence:

```
(run^1 (q) (minusr**o ((⌜3⌝) q) ⌜5⌝))
```

diverges because the call to *appendr*^{*o} keeps succeeding, and because after each success the call to *minusr*^{*o} fails.

Of course not every use of *minusr*^{**o} results in divergence. Let us find ten lists of numbers that contain seven and whose difference is three.

```
(run^10 (q)
  (fresh (x y)
    (≡ (x (⌜7⌝) y) q)
    (minusr**o q ⌜3⌝))) ⇒
(((⌜7⌝) (⌜4⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜4⌝))
 (() (⌜7⌝) (⌜1⌝ ⌜3⌝))
 (() (⌜7⌝) (⌜2⌝ ⌜2⌝))
 (() (⌜7⌝) (⌜4⌝ ⌜0⌝))
 (() (⌜7⌝) (⌜3⌝ ⌜1⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜0⌝ ⌜4⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜1⌝ ⌜3⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜2⌝ ⌜2⌝))
 (() (⌜7⌝) (⌜0⌝ ⌜4⌝ ⌜0⌝)))
```

These values give the impression that x is always associated with the empty list, which is not true. For example, when we replace **run**^{10} with **run**^{70}, then the twenty-third and seventieth values are ((⌜10⌝) (⌜7⌝) ()) and ((⌜11⌝) (⌜7⌝) (⌜1⌝)), respectively.

Let us exclude values in which sublists contain zeros, and display the concatenation of the sublists to make the results more readable.

```
(run^10 (q)
  (fresh (x y in**)
    (≡ (x (⌜7⌝) y) in**)
    (minusr**o in** ⌜3⌝)
    (appendr*o in** q)
    (positive*o q))) ⇒
((⌜7⌝ ⌜4⌝)
 (⌜7⌝ ⌜1⌝ ⌜3⌝)
 (⌜7⌝ ⌜2⌝ ⌜2⌝)
 (⌜7⌝ ⌜3⌝ ⌜1⌝)
 (⌜7⌝ ⌜1⌝ ⌜1⌝ ⌜2⌝)
 (⌜7⌝ ⌜1⌝ ⌜2⌝ ⌜1⌝)
 (⌜7⌝ ⌜2⌝ ⌜1⌝ ⌜1⌝)
 (⌜10⌝ ⌜7⌝)
 (⌜7⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝ ⌜1⌝)
 (⌜11⌝ ⌜7⌝ ⌜1⌝))
```

Of course there are still infinitely many values, even after filtering out lists containing zeros.

Finally, let us replace the second argument to $minusr^{**o}$ with a fresh variable.

```
(run^10 (q)
  (fresh (x y in* in** out)
    (≡ (in* out) q)
    (≡ (x (⌜7⌝) y) in**)
    (minusr**o in** out)
    (appendr*o in** in*)
    (positive*o in*))) ⇒
```

```
(((⌜7⌝ ⌜1⌝) ⌜6⌝)
 ((⌜7⌝ ⌜2⌝) ⌜5⌝)
 ((⌜7⌝ ⌜3⌝) ⌜4⌝)
 ((⌜7⌝ ⌜4⌝) ⌜3⌝)
 ((⌜7⌝ ⌜5⌝) ⌜2⌝)
 ((⌜7⌝ ⌜6⌝) ⌜1⌝)
 ((⌜7⌝ ⌜7⌝) ⌜0⌝)
 ((⌜7⌝ ⌜1⌝ ⌜1⌝) ⌜5⌝)
 ((⌜7⌝ ⌜1⌝ ⌜2⌝) ⌜4⌝)
 ((⌜7⌝ ⌜2⌝ ⌜1⌝) ⌜4⌝))
```

When we replace $\mathbf{run^{10}}$ with $\mathbf{run^{30}}$, the thirtieth value is

```
((⌜_0+7⌝ ⌜7⌝) _0)
```

This value shows that $(n + 7) - 7 = n$ for every n.

5. Conclusions

We have seen how to define both variadic and pseudo-variadic relations in miniKanren, an embedding of logic programming in Scheme. A variadic relation is defined using Scheme's var-args facility, and can take a variable number of arguments. A pseudo-variadic relation takes two arguments, the first of which is a list or a variable representing a list; the ability to pass a fresh variable as the first argument makes a pseudo-variadic relation more flexible than its variadic equivalent.

Just as certain variadic Scheme functions can be defined using the *foldr* or *foldl* abstractions, certain pseudo-variadic relations can be defined using the $foldr^o$ or $foldl^o$ relational abstractions. For example, pseudo-variadic versions of $plus^o$ that add arbitrarily many numbers can be defined using either relational abstraction. Another example is $subsetl^o$, which is defined using $foldl^o$ but cannot be redefined using $foldr^o$. Even those pseudo-variadic relations that can be defined using both relational abstractions may exhibit different divergence behavior when using one abstraction instead of the other. And some pseudo-variadic relations, such as $minusr^{*o}$, cannot be defined using either $foldr^o$ or $foldl^o$.

We have also seen how to define double-pseudo-variadic relations, which are a generalization of pseudo-variadic relations. A double-pseudo-variadic relation takes two arguments, the first of which is either a list of lists or a variable representing a list of lists. As with pseudo-variadic relations, certain double-pseudo-variadic relations can be defined using relational abstractions—for example, $plusr^{**o}$ is defined using $foldr^{*o}$.

The benefits of using Scheme as the host language for miniKanren are demonstrated by the ease with which we can define higher-order relational abstractions such as $foldr^{*o}$. We hope the examples we have presented inspire the reader to experiment with miniKanren, and to experience the fun of combining relational programming with Scheme.

Acknowledgments

The implementation described in the appendix was developed with Oleg Kiselyov and Chung-chieh Shan. We are also grateful to Michael Adams and Kyle Ross for their comments on our earlier proposals for pseudo-variadic relations. We thank Ronald Garcia, Oleg Kiselyov, Chung-chieh Shan, Michael Adams, and the anonymous reviewers for their detailed and insightful comments on drafts of this paper. We have found Dorai Sitaram's excellent SLaTeX package invaluable for typesetting our code. Finally, we thank Mitchell Wand for his encouragement and clever last-minute tweaking of the implementation.

References

[1] A declarative applicative logic programming system. http://kanren.sourceforge.net/.

[2] Baader, F., and Snyder, W. Unification theory. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds., vol. I. Elsevier Science, 2001, ch. 8, pp. 445–532.

[3] Clocksin, W. F. *Clause and Effect: Prolog Programming and the Working Programmer*. Springer, 1997.

[4] Colmerauer, A. An introduction to Prolog III. *Commun. ACM 33*, 7 (1990), 69–90.

[5] Edelstein, H. add-up problem. Message posted on newsgroup comp.lang.scheme on July 5, 2006 07:12 EST.

[6] Friedman, D. P., Byrd, W. E., and Kiselyov, O. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

[7] Henderson, F., Conway, T., Somogyi, Z., and Jeffery, D. The Mercury language reference manual. Tech. Rep. 96/10, University of Melbourne, 1996.

[8] Jones, S. L. P. Haskell 98: Standard prelude. *J. Funct. Program. 13*, 1 (2003), 103–124.

[9] Kelsey, R., Clinger, W., and Rees, J. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76.

[10] Kiselyov, O., Shan, C., Friedman, D. P., and Sabry, A. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005* (2005), O. Danvy and B. C. Pierce, Eds., ACM, pp. 192–203.

[11] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991), 55–92.

[12] Naish, L. Pruning in logic programming. Tech. Rep. 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.

[13] Naish, L. Higher-order logic programming in Prolog. Tech. Rep. 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, Feb. 1996.

[14] O'Keefe, R. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.

[15] Sitaram, D. Programming in Schelog. http://www.ccs.neu.edu/home/dorai/schelog/schelog.html.

[16] Spivey, J. M., and Seres, S. Combinators for logic programming. In *The Fun of Programming*, J. Gibbons and O. de Moor, Eds. Palgrave, 2003, ch. 9, pp. 177–200.

[17] Wadler, P. The essence of functional programming. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, Jan., 1992), ACM Press, pp. 1–14.

A. miniKanren Implementation

miniKanren evolved from Kanren [1]; its implementation comprises three kinds of operators: functions such as *unify* and *reify*, which take substitutions explicitly; goal constructors ≡, **cond**^{e}, and **fresh**, which take substitutions implicitly; and the interface operator **run**. We represent substitutions as association lists associating variables with values.

unify is based on the triangular model of substitutions (See Baader and Snyder [2], for example.). Vectors should not occur in arguments passed to *unify*, since we represent variables as vectors.

```
(define unify
  (lambda (u v s)
    (let ((u (walk u s))
          (v (walk v s)))
      (cond
        ((eq? u v) s)
        ((var? u)
         (cond
           ((var? v) (ext-s u v s))
           (else (ext-s✓ u v s))))
        ((var? v) (ext-s✓ v u s))
        ((and (pair? u) (pair? v))
         (let ((s (unify (car u) (car v) s)))
           (and s (unify (cdr u) (cdr v) s))))
        ((equal? u v) s)
        (else #f)))))

(define walk
  (lambda (v s)
    (cond
      ((var? v)
       (let ((a (assq v s)))
         (cond
           (a (walk (cdr a) s))
           (else v))))
      (else v))))

(define ext-s✓
  (lambda (x v s)
    (cond
      ((occurs✓ x v s) #f)
      (else (ext-s x v s)))))

(define occurs✓
  (lambda (x v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (eq? v x))
        ((pair? v)
         (or (occurs✓ x (car v) s) (occurs✓ x (cdr v) s)))
        (else #f)))))

(define ext-s
  (lambda (x v s)
    (cons (x . v) s)))

(define empty-s ())

(define var vector)

(define var? vector?)
```

reify takes a substitution and an arbitrary value, perhaps containing variables. *reify* first uses *walk** to apply the substitution to a value and then methodically replaces any variables with reified names.

```
(define reify
  (letrec
    ((reify-s
       (lambda (v s)
         (let ((v (walk v s)))
           (cond
             ((var? v) (ext-s v (reify-name (length s)) s))
             ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
             (else s))))))
    (lambda (v s)
      (let ((v (walk* v s)))
        (walk* v (reify-s v empty-s))))))

(define walk*
  (lambda (w s)
    (let ((v (walk w s)))
      (cond
        ((var? v) v)
        ((pair? v) (cons (walk* (car v) s) (walk* (cdr v) s)))
        (else v)))))

(define reify-name
  (lambda (n)
    (string→symbol
      (string-append "_" "." (number→string n)))))
```

A goal g is a function that maps a substitution s to an ordered sequence of zero or more values—these values are almost always substitutions. (For clarity, we notate **lambda** as λ_{G} when creating such a function g.) Because the sequence of values may be infinite, we represent it not as a list but as a special kind of stream, a^∞.

Such streams contain either zero, one, or more values [10, 16]. We use (**mzero**) to represent the empty stream of values. If a is a value, then (**unit** a) represents the stream containing just a. To represent a non-empty stream we use (**choice** a f), where a is the first value in the stream, and where f is a function of zero arguments. (For clarity, we notate **lambda** as λ_{F} when creating such a function f.) Invoking the function f produces the remainder of the stream. (**unit** a) can be represented as (**choice** a (λ_{F} () (**mzero**))), but the **unit** constructor avoids the cost of building and taking apart pairs and invoking functions, since many goals return only singleton streams. To represent an incomplete stream, we create an f using (**inc** e), where e is an *expression* that evaluates to an a^∞.

```
(define-syntax mzero
  (syntax-rules ()
    ((_) #f)))

(define-syntax unit
  (syntax-rules ()
    ((_ a) a)))

(define-syntax choice
  (syntax-rules ()
    ((_ a f) (cons a f))))

(define-syntax inc
  (syntax-rules ()
    ((_ e) (λF () e))))
```

To ensure that streams produced by these four a^∞ constructors can be distinguished, we assume that a singleton a^∞ is never **#f**, a function, or a pair whose *cdr* is a function. To discriminate among these four cases, we define **case**$^\infty$.

```
(define-syntax case^∞
  (syntax-rules ()
    ((_ e on-zero ((â) on-one) ((a f) on-choice) ((f̂) on-inc))
     (let ((a^∞ e))
       (cond
         ((not a^∞) on-zero)
         ((procedure? a^∞) (let ((f̂ a^∞)) on-inc))
         ((and (pair? a^∞) (procedure? (cdr a^∞)))
          (let ((a (car a^∞)) (f (cdr a^∞))) on-choice))
         (else (let ((â a^∞)) on-one)))))))
```

The simplest goal constructor is ≡, which returns either a singleton stream or an empty stream, depending on whether the arguments unify with the implicit substitution. As with the other goal constructors, ≡ always expands to a goal, even if an argument diverges. We avoid the use of **unit** and **mzero** in the definition of ≡, since *unify* returns either a substitution (a singleton stream) or **#f** (our representation of the empty stream).

```
(define-syntax ≡
  (syntax-rules ()
    ((_ u v)
     (λG (s)
       (unify u v s)))))
```

cond^e is a goal constructor that combines successive **cond**^{e}-clauses using **mplus***. To avoid unwanted divergence, we treat the **cond**^{e}-clauses as a single **inc** stream. Also, we use the same implicit substitution for each **cond**^{e}-clause. **mplus*** relies on *mplus*, which takes an a^∞ and an f and combines them (a kind of *append*). Using **inc**, however, allows an argument to *become* a stream, thus leading to a relative fairness because all of the stream values will be interleaved.

```
(define-syntax cond^e
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (s)
       (inc
         (mplus*
           (bind* (g0 s) g ...)
           (bind* (g1 s) ĝ ...) ...))))))

(define-syntax mplus*
  (syntax-rules ()
    ((_ e) e)
    ((_ e0 e ...) (mplus e0 (λF () (mplus* e ...))))))

(define mplus
  (lambda (a^∞ f)
    (case^∞ a^∞
      (f)
      ((a) (choice a f))
      ((a f̂) (choice a (λF () (mplus (f̂) f))))
      ((f̂) (inc (mplus (f) f̂))))))
```

If the body of **cond**^{e} were just the **mplus*** expression, then the **inc** clauses of *mplus*, *bind*, and *take* would never be reached, and there would be no interleaving of values.

fresh is a goal constructor that first lexically binds its variables (created by *var*) and then, using **bind***, combines successive goals. **bind*** is short-circuiting: since the empty stream (**mzero**) is represented by **#f**, any failed goal causes **bind*** to immediately return **#f**. **bind*** relies on *bind* [11, 17], which applies the goal g to each element in a^∞. These a^∞'s are then merged together with *mplus* yielding an a^∞. (*bind* is similar to Lisp's *mapcan*, with the arguments reversed.)

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (s)
       (let ((x (var 'x)) ...)
         (bind* (g0 s) g ...))))))

(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
    ((_ e g0 g ...)
     (let ((a^∞ e))
       (and a^∞ (bind* (bind a^∞ g0) g ...))))))

(define bind
  (lambda (a^∞ g)
    (case^∞ a^∞
      (mzero)
      ((a) (g a))
      ((a f) (mplus (g a) (λF () (bind (f) g))))
      ((f) (inc (bind (f) g))))))
```

To minimize heap allocation we create a single λ_G closure for each goal constructor, and we define **bind*** and **mplus*** to manage sequences, not lists, of goal-like expressions.

run, and therefore *take*, converts an f to a list. We wrap the result of (*reify x s*) in a list so that the **case**$^\infty$ in *take* can distinguish a singleton a^∞ from the other three a^∞ types. We could simplify **run** by using *var* to create the fresh variable x, but we prefer that **fresh** be the only operator that calls *var*.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (λF ()
         (let ((ĝ (fresh (x)
                    (λG (s)
                      (bind* (g0 s) g ...
                        (λG (s)
                          (list (reify x s)))))))))
           (ĝ empty-s)))))))

(define take
  (lambda (n f)
    (if (and n (zero? n))
        '()
        (case^∞ (f)
          '()
          ((a) a)
          ((a f) (cons (car a) (take (and n (- n 1)) f)))
          ((f) (take n f))))))
```

If the first argument to *take* is **#f**, we get the behavior of **run***. It is trivial to write a read-eval-print loop that uses the **run*** interface by redefining *take*.

This ends the implementation of the subset of miniKanren used in this paper. Below we define the three additional goal constructors that complete the entire embedding: **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program, and **project**, which can be used to access the values of variables.

cond^a and **cond**^{u} correspond to the *committed-choice* of Mercury, and are used in place of Prolog's *cut* [7, 12]. Unlike **cond**^{e}, only one **cond**^{a}-clause or **cond**^{u}-clause can return an a^∞: the first clause whose first goal succeeds. With **cond**^{a}, the entire stream returned by the first goal is passed to **bind*** (see **pick**^{a}). With **cond**^{u}, a singleton stream is passed to **bind***—this stream contains the first value of the stream returned by the first goal (see **pick**^{u}). The examples from chapter 10 of *The Reasoned Schemer* [6] demonstrate how **cond**^{a} and **cond**^{u} can be useful and the pitfalls that await the unsuspecting reader.

```
(define-syntax cond^a
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (s)
       (if* (pick^a (g0 s) g ...) (pick^a (g1 s) ĝ ...) ...)))))

(define-syntax cond^u
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (s)
       (if* (pick^u (g0 s) g ...) (pick^u (g1 s) ĝ ...) ...)))))

(define-syntax if*
  (syntax-rules ()
    ((_) (mzero))
    ((_ (pick e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (if* b ...)
         ((a) (bind* a^∞ g ...))
         ((a f) (bind* (pick a a^∞) g ...))
         ((f) (inc (loop (f)))))))))

(define-syntax pick^a
  (syntax-rules ()
    ((_ a a^∞) a^∞)))

(define-syntax pick^u
  (syntax-rules ()
    ((_ a a^∞) (unit a))))
```

project applies the implicit substitution to zero or more lexical variables, rebinds those variables to the values returned, and then evaluates the goal expressions in its body. The body of a **project** typically includes at least one **begin** expression—any expression is a goal expression if its value is a miniKanren goal. **project** has many uses, such as displaying the values associated with variables when tracing a program.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (s)
       (let ((x (walk* x s)) ...)
         (bind* (g0 s) g ...))))))
```

A Self-Hosting Evaluator using HOAS

A Scheme Pearl

Eli Barzilay
Northeastern University
eli@barzilay.org

Abstract

We demonstrate a tiny, yet non-trivial evaluator that is powerful enough to run practical code, including itself. This is made possible using a Higher-Order Abstract Syntax (HOAS) representation — a technique that has become popular in syntax-related research during the past decade. With a HOAS encoding, we use functions to encode binders in syntax values, leading to an advantages of reflecting binders rather than re-implementing them.

In Scheme, hygienic macros cover problems that are associated with binders in an elegant way, but only when extending the language, i.e., when we work at the meta-level. In contrast, HOAS is a useful object-level technique, used when we need to *represent* syntax values that contain bindings — and this is achieved in a way that is simple, robust, and efficient. We gradually develop the code, explaining the technique and its benefits, while playing with the evaluator.

1. Introduction

Higher-Order Abstract Syntax (HOAS) is a technique for representing syntax with bindings using functions. This is a form of reflection in the sense that binders in the object level are represented using binders in the meta level. The result is simple (no need for sophisticated substitution mechanisms), robust (the meta-level is our language, which better implement scope correctly), and efficient (as it is part of the core implementation).

HOAS has been in use for a while now [13], a good overview is given by Hofmann [9], and in [1, Section 4.7]. However, it is more popular in the strictly typed world than it is in Scheme. In part, this may be due to Scheme's hygienic macro facility, which allows hooking new kinds of syntactic constructs into the language in a way that respects lexical scope. Using a high level macro system means that Schemers rarely need to represent syntax directly — instead, they work at the meta-level, extending the language itself. This is unfortunate, as HOAS can be a useful tool for syntax-related work. Furthermore, it is possible to formalize HOAS in a way that is more natural in a dynamically-typed language than it is in statically-typed ones, which corresponds to a HOAS formalization in Nuprl [3] that builds on the predicative nature of its type theory [2, 1, 12].

The purpose of this Scheme pearl is to demonstrate the use of HOAS in Scheme, with the goal of making this technique more accessible to Schemers[1]. As demonstrated below, using macros in Scheme facilitates the use of HOAS, as there is no need for an external tool for translating concrete syntax into HOAS representations. In itself, HOAS is a representation tool for object-level values, not for meta-level work where bindings are directly accessible in some way. It is, however, used in some meta-linguistic systems for implementing syntactic tools[2]. For example, it could be used as the underlying term representation in a language-manipulating tool like PLT's Reduction Semantics [11].

2. A Toy Evaluator

The presentation begins with a simple evaluator. Our goal is to evaluate a Lambda-Calculus-like language using reductions, so we need a representation for `lambda` abstractions and applications. To make this example more practical, we also throw in a conditional `if` special form, make it handle multiple arguments, and use call-by-value. A common evaluator sketch for such a language is[3]:

```
(define (ev expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev (if (ev (cadr expr)) (caddr expr) (cadddr expr)))]
        [else (ev (let ([f (ev (car expr))])
                    (substitute (body-of f)
                                (args-of f)
                                (map ev (cdr expr)))))]))
```

where an application is always assumed to have an abstraction in its head position, and the `args-of` and `body-of` functions pull out the corresponding parts. As expected, the main issue here is implementing a proper substitution function. Common approaches include using symbols and renaming when needed, or using symbols 'enriched' with lexical binding information ('colors').

[1] The presentation is loosely based on a comp.lang.scheme post from October 2002.

[2] It might be possible that HOAS can be used for implementing a low-level macro facility that the high-level hygienic macro system builds on. HOAS should not be confused with current low-level syntactic systems like syntactic closures or `syntax-case`.

[3] Note that details like error checking are omitted, and that we use atomic Scheme values such as booleans and numbers to represent themselves.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

On the other hand, we can use a higher-order abstract syntax representation for our language, which will make things easier to handle than raw S-expressions. For this, we represent an abstraction using a Scheme function, and bound occurrences are represented as bound occurrences in the Scheme code. For example,

```
(lambda (x y) (if x x y))
```

is represented by

```
(lambda (x y) (list 'if x x y))
```

— and substitution is free as it is achieved by applying the *Scheme* function.

3. Creating HOAS Representations

Given that our representation of `lambda` abstractions uses Scheme `lambda` expressions, we need some facility to create such representation while avoiding the possible confusion when using `lambda` for different purposes. The common approach for creating such representations is to use a preprocessor that translates concrete syntax into a HOAS value. In Scheme, this is easily achieved by a macro that transforms its body to the corresponding representation:

```
;; Translates simple terms into a HOAS representation
;; values, which are:
;; Term = atom                       ; literals
;;      | (list 'if Term Term Term)  ; conditionals
;;      | (Term ... -> Term)         ; abstractions
;;      | (list Term ...)            ; applications
(define-syntax Q
  (syntax-rules (lambda if)
    [(Q (lambda args b)) (lambda args (Q b))]
    [(Q (if x y z))      (list 'if (Q x) (Q y) (Q z))]
    [(Q (f x ...))       (list (Q f) (Q x) ...)]
    [(Q x)               x]))
```

A few interaction examples can help clarify the nature of these values:

```
> (Q 1)
1
> (Q (+ 1 2))
(#<primitive:+> 1 2)
> (Q (if 1 2 3))
(if 1 2 3)
> (Q (lambda (x) (+ x 1)))
#<procedure>
```

The last one is important — the `lambda` expression is represented by a Scheme procedure, which, when applied, returns the result of substituting values for bound identifiers:

```
> (define foo (Q (lambda (x) (+ x 1))))
> (foo 2)
(#<primitive:+> 2 1)
```

Using the representations that the 'quoting' macro `Q` creates, a complete substitution-based evaluator is easily written:

```
;; ev : Term -> Val
;; evaluate an input term into a Scheme value
(define (ev expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev (if (ev (cadr expr)) (caddr expr) (cadddr expr)))]
        [else (ev (apply (ev (car expr))
                         (map ev (cdr expr))))]))
```

Note that the underlined `apply` expression invokes the Scheme procedure which performs the substitution that is needed for the beta-reduction: it is a '`Term`→`Term`' function. The result of this application expression is therefore a piece of (post-substitution) syntax that requires further evaluation.

In addition, this is a simple substitution-based evaluator — no environments, identifier lookups, or mutation. Scheme values are used as self-evaluating literals (some achieved by a Scheme identifier reference), including Scheme procedures that are exposed as primitive values. Specifically, the last `cond` clause is used for both primitive function applications and beta reductions — this leads to certain limitations and possible errors, so it is fixed below.

It is easy to confuse the current representation as a trick; after all, we represent abstractions using Scheme abstractions, and beta-reduce using Scheme applications — sounds like we end up with a simple meta-circular Scheme evaluator that inherits Scheme's features. This is not the case, however: Scheme applications achieves nothing more than substitution. To demonstrate this, the evaluator can easily be changed to use a lazy evaluation regimen if it avoids evaluating abstraction arguments. This requires a distinction between strict and non-strict positions. For simplicity, we only distinguish primitive functions (all arguments are strict) and abstractions (no arguments are strict) using MzScheme's `primitive?`[4] predicate:

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value, lazy version
(define (ev* expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev* ((if (ev* (cadr expr)) caddr cadddr) expr))]
        [else (ev* (let ([f (ev* (car expr))])
                     (apply f (if (primitive? f)
                                  (map ev* (cdr expr))
                                  (cdr expr)))))]))
```

And the result is a lazy language, where we can even use the call-by-name fixpoint combinator:

```
> (ev (Q ((lambda (x y z) (if x y z))
          #t (display "true\n") (display "false\n"))))
true
false
> (ev* (Q ((lambda (x y z) (if x y z))
           #t (display "true\n") (display "false\n"))))
true
> (ev* (Q (((lambda (f)
              ((lambda (x) (f (x x)))
               (lambda (x) (f (x x)))))
            (lambda (fact)
              (lambda (n)
                (if (zero? n) 1 (* n (fact (- n 1)))))))
           5)))
120
```

4. Advantages of the HOAS Representation

At this point we can see the advantages of the HOAS representation. These are all due to the fact that the Scheme binding mechanism is *reflected* rather than *re-implemented*.

Free substitution: since we use Scheme functions, the Scheme implementation provides us with free substitution — we get a substituting evaluator, without the hassle of implementing substitution.

Robust: dealing with the subtleties of identifier scope (substitution, alpha renaming, etc) is usually an error-prone yet critical element in code that deals with syntax. In our evaluator, we

[4] A predicate that identifies primitive built-in procedures.

need not worry about these issues, since it reflects the mechanism that already exists in the implementation we use.

Efficient: the representation lends itself to efficient substitution for two reasons. First, function calls are an essential part of functional languages that must be very efficient; a feature that our substitution inherits. Second, if we incrementally substitute some syntax value with multiple binding levels, then the substitutions are not carried out immediately but pushed to substitution cache contexts (=environments), which are the implementation's efficient representation of closures.

Good integration: representing concrete syntax with Scheme S-expressions is superior to the flat string representations that is found in other languages because the structure of the syntax is reflected in syntax values (values are "pre-parsed" into trees). In a similar way, HOAS adds yet another dimension to the representation — scope is an inherent part of representations (lexical scopes are already identified and turned to closures). We therefore enjoy all functionality that is related to scope in our implementation. For example, the unbound identifiers are caught by the implementation, analysis tools such as DrScheme's "Check Syntax" [6] work for bindings in the representation, macros can be used, etc.

These advantages, however, do not come without a price. More on this below.

5. Improving the Code

So far, the evaluator is simple, but the code is somewhat messy: `lambda` abstractions and primitive procedures are conflated, lists are used both as values and as syntax representations. Furthermore, the evaluator is not as complete as it needs to be to host itself. We begin by improving the code, and in the following section we will extend it so it can run itself.

We begin by introducing a new type for our syntax objects, and use this type to create tagged values for lambda abstractions and applications:

```
;; A type for syntax representation values
;; Term = atom                              ; literals
;;      | (term 'if Term Term Term)         ; conditionals
;;      | (term 'lam (Term ... -> Term))    ; abstractions
;;      | (term 'app Term ...)              ; applications
(define-struct term (tag exprs) #f)
(define (term tag . args) (make-term tag args))

;; Translates simple terms into a HOAS representation
(define-syntax Q
  (syntax-rules (lambda if)
    [(Q (lambda args b)) (term 'lam (lambda args (Q b)))]
    [(Q (if x y z))      (term 'if  (Q x) (Q y) (Q z))]
    [(Q (f x ...))       (term 'app (Q f) (Q x) ...)]
    [(Q x)               x]))
```

`ev` is then adapted to process values of this type.

```
;; ev : Term -> Val
;; evaluate an input term into a Scheme value
(define (ev expr)
  (if (term? expr)
    (let ([subs (term-exprs expr)])
      (case (term-tag expr)
        [(lam) expr]
        [(if)  (ev ((if (ev (car subs)) cadr caddr) subs))]
        [(app) (let ([f (ev (car subs))]
                     [args (map ev (cdr subs))])
                 (cond [(and (term? f) (eq? 'lam (term-tag f)))
                        (ev (apply (car (term-exprs f)) args))]
                       [(procedure? f)
                        (apply f args)]
                       [else (error 'ev "bad procedure")]))]
        [else (error 'ev "bad tag")]))
    expr))
```

We can now test this evaluation procedure:

```
> (ev (Q (lambda (x) (+ x 1))))
#3(struct:term lam (#<procedure>))
> (ev (Q ((lambda (x) (+ x 1)) 2)))
3
> (define plus1 (Q (lambda (x) (+ x 1))))
> (ev (Q (plus1 2)))
3
```

As the previous version, this evaluator does not maintain its own environment, instead, it uses the Scheme environment (in cooperation with the quotation macro that leaves bindings untouched). This is used as a definition mechanism that is demonstrated in the last example — but we have to be careful to use such values only in a syntax-representation context. Because the representation is using Scheme closures, we can use recursive Scheme definitions to achieve recursion in our interpreted language:

```
> (define fact
    (Q (lambda (n)
         (if (zero? n) 1 (* n (fact (- n 1)))))))
> (ev (Q (fact 30)))
265252859812191058636308480000000
```

Again, making this evaluator lazy is easy: we only need to avoid evaluating the arguments on beta reductions. In fact, we can go further and 'compile' `lambda` expressions into Scheme closures that will do the reduction and use `ev*` to continue evaluating the result. To deal with strict primitives properly, we evaluate them to a wrapper function that evaluates its inputs[5]:

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value,
;; this version is lazy, and 'compiles' closures to Scheme procedures
(define (ev* expr)
  (cond
    [(term? expr)
     (let ([subs (term-exprs expr)])
       (case (term-tag expr)
         [(lam) (lambda args (ev* (apply (car subs) args)))]
         [(if)  (ev* ((if (ev* (car subs)) cadr caddr) subs))]
         [(app) (apply (ev* (car subs)) (cdr subs))]
         [else (error 'ev "bad tag")]))]
    [(primitive? expr)
     (lambda args (apply expr (map ev* args)))]
    [else expr]))
```

On first look, this change seems a bit dangerous — not only is a `lambda` expression represented as a Scheme closure, evaluating it returns a Scheme closure. In fact, this approach works as the types demonstrate: the function that is part of the representation is '`Term`→`Term`', whereas the 'compiled' closure is a '`Term`→`Val`' function. Note also that Scheme primitives act as primitives of the interpreted language ('`Val`→`Val`'), and the evaluator wraps them as a '`Term`→`Val`' function that allows uniform treatment of both cases in applications.

Here are a few examples to compare with the previous evaluator:

[5] Ideally, *any* procedure that is not the result of evaluating a `lambda` expression should be wrapped. In MzScheme it is possible to tag some closures using applicable structs, but in this paper the code is kept short.

```
> (ev* (Q (lambda (x) (+ x 1))))
#<procedure>
> (ev* (Q ((lambda (x) (+ x 1)) 2)))
3
> (ev* (Q ((lambda (x y) (+ x 1)) 2 (/ 2 0))))
3
> ((ev* (Q (lambda (x) (+ x 1))))
   (Q 2))
3
```

In the last example, the '`Term→Val`' procedure that is the result of evaluating the first part is directly applied on `(Q 2)` (a syntax) which is essentially how the outermost application of the second example is handled. This application jumps back into the evaluator and continues the computation.

Using the Scheme toplevel for definitions, we can define and use a lazy fixpoint combinator:

```
> (define Y
    (ev* (Q (lambda (f)
              ((lambda (x) (f (x x)))
               (lambda (x) (f (x x))))))))
> (define fact0
    (ev* (Q (lambda (fact)
              (lambda (n)
                (if (zero? n) 1 (* n (fact (- n 1)))))))))
> (ev* (Q (Y fact0)))
#<procedure>> (ev* (Q ((Y fact0) 30)))
265252859812191058636308480000000
```

Finally, as an interesting by-product of this namespace sharing, we can even use the call-by-name fixpoint combinator with Scheme code, as long as we use `ev*` to translate the function into a Scheme function:

```
> ((Y (lambda (fact)
        (lambda (n)
          (let ([fact (ev* fact)])
            (if (zero? n) 1 (* n (fact (- n 1))))))))
   30)
265252859812191058636308480000000
```

6. Making `ev` Self-Hosting

In preparation for making our evaluator self-hosting, we need to deal with representations of all forms that are used in its definition. Again, to make things simple, we avoid adding new core forms — instead, we translate the various forms to ones that the evaluator already knows how to deal with. We will use 'nested' instantiations of our evaluator, which will require nested use of the `Q` quotation form — this is a minor complication that could be solved using macro-CPS [8, 10], but in MzScheme [7] it is easier to write a `syntax-case`-based macro, which uses a simple loop for nested occurrences of `Q`. The new (and final) definition is in Figure 1. On first look it seems complex, but it is merely translating additional forms into known ones, and propagates the transformation into the `delay` special form (which will be needed shortly).

The lazy evaluator is slightly modified: call-by-name is too slow to be usable when nesting multiple evaluators, so we change it to use call-by-need instead. For this, we make it create `ev*` promises for function arguments, and automatically force promise values so they are equivalent to plain values. We also need to treat the `term` constructor as a primitive (otherwise it will contain promises instead of values). The definition follows.

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value, uses call-by-need
(define (ev* expr)
  (cond
    [(term? expr)
     (let ([subs (term-exprs expr)])
       (case (term-tag expr)
         [(lam) (lambda args
                  (ev* (apply (car subs)
                              (map (lambda (a) (delay (ev* a)))
                                   args))))]
         [(if)  (ev* ((if (ev* (car subs)) cadr caddr) subs))]
         [(app) (apply (ev* (car subs)) (cdr subs))]
         [else (error 'ev "bad tag")]))]
    [(promise? expr) (ev* (force expr))]
    [(primitive*? expr)
     (lambda args (apply expr (map ev* args)))]
    [else expr]))

(define (primitive*? x)
  (or (primitive? x) (eq? x term)))
```

And with this change we are finally ready to run the evaluator code in itself.

7. Bootstrapping the Evaluator

First, we use the strict evaluator to evaluate a nested copy of itself. In this definition, `ev` is the same as the strict version above, and its code is used with no change.

```
(define ev1
  (ev* (Q (Y (lambda (ev)
               (lambda (expr)
                 (cond
                   [(term? expr)
                    (let ([subs (term-exprs expr)])
                      (case (term-tag expr)
                        [(lam) (lambda args
                                 (ev (apply (car subs) args)))]
                        [(if)  (ev ((if (ev (car subs))
                                        cadr caddr)
                                    subs))]
                        [(app) (apply (ev (car subs))
                                      (map ev (cdr subs)))]
                        [else (error 'ev1 "bad tag")]))]
                   [else expr])))))))
```

We can verify that this evaluator works as expected:

```
> (ev (Q (ev1 (Q (+ 1 2)))))
3
> (ev (Q (ev1 (Q ((lambda (x) (+ x 2)) 1)))))
3
```

We can continue this and implement a third evaluator in `ev1` — using the same definition once again. The result is again working fine.

```
> (define ev2
    (ev (Q (ev1 (Q (lambda (expr)
                     ;; Same code as ev1, substituting 'ev2' for 'ev1'
                     ))))))
> (ev (Q (ev1 (Q (ev2 (Q (+ 1 2)))))))
3
> (ev (Q (ev1 (Q (ev2 (Q ((lambda (x) (+ x 2)) 1)))))))
3
```

It is interesting to compare the performance of the three evaluators. We do this by defining a Fibonacci function in each of the three levels and in Scheme:

```
(define fib
  (lambda (n)
    (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))))
```

```
;; Translates terms into a HOAS representation
(define-syntax (Q s)
  (let transform ([s s])
    (syntax-case s (Q quote lambda if let and or cond case else delay)
      [(Q (Q x)) ; transform once, then reprocess:
       (with-syntax ([1st-pass (transform (syntax (Q x)))])
         (syntax (Q 1st-pass)))]
      [(Q (quote x))           (syntax 'x)]
      [(Q (lambda args b))     (syntax (term 'lam (lambda args (Q b))))]
      [(Q (if x y z))          (syntax (term 'if  (Q x) (Q y) (Q z)))]
      [(Q (let ([x v] ...) b)) (syntax (Q ((lambda (x ...) b) v ...)))]
      [(Q (and))               (syntax #t)]
      [(Q (and x))             (syntax x)]
      [(Q (and x y ...))       (syntax (Q (if x (and y ...) #f)))]
      [(Q (or))                (syntax #f)]
      [(Q (or x))              (syntax x)]
      [(Q (or x y ...))        (syntax (Q (let ([x* x]) (if x* x* (or y ...)))))]
      [(Q (cond))              (syntax 'unspecified)]
      [(Q (cond [else b]))     (syntax (Q b))]
      [(Q (cond [test b] clause ...))
                               (syntax (Q (if test b (cond clause ...))))]
      [(Q (case v))            (syntax 'unspecified)]
      [(Q (case v [else b]))   (syntax (Q b))]
      [(Q (case v [(tag) b] clause ...)) ; (naive translation)
                               (syntax (Q (if (eqv? v 'tag) b (case v clause ...))))]
      [(Q (delay x))           (syntax (delay (Q x)))]
      [(Q (f x ...))           (syntax (term 'app (Q f) (Q x) ...))]
      [(Q x)                   (syntax x)])))
```

Figure 1. Full quotation code

```
(define fib0
  (ev (Q (lambda (n) ...))))
(define fib1
  (ev (Q (ev1 (Q (lambda (n) ...))))))
(define fib2
  (ev (Q (ev1 (Q (ev2 (Q (lambda (n) ...)))))))))
```

Measuring their run-time shows the expected blowup with each layer of representation, and that even at three levels of nesting it is still usable.

```
> (time (fib 18))
cpu time: 1 real time: 1 gc time: 0
2584
> (time (ev (Q (fib0 18))))
cpu time: 105 real time: 133 gc time: 65
2584
> (time (ev (Q (ev1 (Q (fib1 18))))))
cpu time: 618 real time: 637 gc time: 394
2584
> (time (ev (Q (ev1 (Q (ev2 (Q (fib2 18))))))))
cpu time: 3951 real time: 4131 gc time: 2612
2584
```

To make things more interesting, we can try variations on this theme. For example, we can nest a strict evaluator in the lazy one, and use the `Y` combinator to get recursion:

```
(define ev*1
  (ev* (Q (Y (lambda (ev)
               (lambda (expr)
                 (cond
                   [(term? expr)
                    (let ([subs (term-exprs expr)])
                      (case (term-tag expr)
                        [(lam) (lambda args
                                 (ev (apply (car subs) args)))]
                        [(if)  (ev ((if (ev (car subs))
                                        cadr caddr)
                                    subs))]
                        [(app) (apply (ev (car subs))
                                      (map ev (cdr subs)))]
                        [else (error 'ev*1 "bad tag")]))]
                   [else expr])))))))
```

The definition of this evaluator is not really strict. In fact, it does not enforce any evaluation strategy — it just inherits it from the language it is implemented in. In this case, this definition is running in `ev*`'s lazy context, which makes the resulting language lazy as well:

```
> (ev* (Q (ev*1 (Q (+ 1 2)))))
3
> (ev* (Q (ev*1 (Q ((lambda (x y) y) (+ 1 "2") 333)))))
333
```

Again, we can repeat this definition to get a third level, then measure the performance of the three levels using `fib` definitions:

```
> (time (ev* (Q (fib*0 18))))
cpu time: 198 real time: 227 gc time: 129
2584
> (time (ev* (Q (ev*1 (Q (fib*1 18))))))
cpu time: 575 real time: 589 gc time: 357
2584
> (time (ev* (Q (ev*1 (Q (ev*2 (Q (fib*2 18))))))))
cpu time: 1186 real time: 1266 gc time: 780
2584
```

It is interesting to note that the blowup factor is much smaller than in the `ev` case. The conclusion is still the same: each evaluator layer increases run-time, but the blowup is small enough to still be practical. (E.g., it is a feasible strategy for implementing DSLs.)

8. HOAS Disadvantages

As mentioned above, HOAS does not come without a price. The two major problems with HOAS representations are well-known:

Exotic terms: we have seen that the functions that are used in HOAS representations are syntactic '`Term`→`Term`' transformers. However, not all of these functions are proper representations — some are not a quotation of any concrete syntax. For example, we can manually construct the following term, which does hold a '`Term`→`Term`' function:

```
(term 'lam
      (lambda (x)
        (if (equal? x (Q 1)) (Q 2) (Q 3))))
```

but it is not a valid representation — there is no `lambda` term that has this as its quotation. Briefly, the problem is that the function is trying to *inspect* its values (it would have been a valid representation had the whole `if` expression been quoted).

This means that we should not allow arbitrary functions into the representation; indeed, major research efforts went into formalizing types in various ways from permutations-based approaches [15, 16, 17], to modal logic [4, 5] and category theory [14]. In [1], another formalism is presented which is of particular interest in the context of Scheme. It relies on a predicative logic system, which corresponds to certain dynamic run-time checks that can exclude formation of exotic terms. This formalism is extended in [12].

Induction: another common problem is that the representation contains functions (which puts a function type in a negative position), and therefore does not easily lend itself to induction. Several solutions for this problem exist. As previously demonstrated [1], these functions behave in a way that makes them directly correspond to concrete syntax. In Scheme, the quotation macro can add the missing information — add the syntactic information that contains enough hints to recover the structure (but see [1] for why this is not straightforward).

As a side note, it is clear that using HOAS is very different in its nature than using high-level Scheme macros. Instead of plain pattern matching and template filling, we need to know and encode the exact lexical structure of any binding form that we wish to encode. Clearly, the code that is presented here is simplified as it has just one binding form, but we would face such problems if we would represent more binding constructs like `let`, `let*` and `letrec`. This is not necessarily a negative feature, since lexical scope needs to be specified in any case.

9. Conclusion

We have presented code that uses HOAS techniques in Scheme. The technique is powerful enough to make it possible to write a small evaluator that can evaluate itself, and — as we have shown — be powerful enough to model different evaluation approaches. In addition to being robust, the encoding is efficient enough to be practical even at three levels of nested evaluators, or when using lazy semantics. HOAS is therefore a useful tool in the Scheme world in addition to the usual host of meta-level syntactic tools.

We plan on further work in this area, specifically, it is possible that using a HOAS representation for PLT's Reduction Semantics [11] tool will result in a speed boost, and a cleaner solution to its custom substitution specification language. Given that we're using Scheme bindings to represent bindings may make it possible to use HOAS-based techniques *combined* with Scheme macros. For example, we can represent a language in Scheme using Scheme binders, allowing it to be extended via Scheme macros in a way that still respects Scheme's lexical scope rules.

References

[1] Eli Barzilay. *Implementing Direct Reflection in Nuprl*. PhD thesis, Cornell University, Ithaca, NY, January 2006.

[2] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[3] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[4] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96*, pages 258–270. ACM Press, New York, 1996.

[5] Joëlle Despeyroux and Pierre Leleu. A modal lambda-calculus with iteration and case constructs. In *Proc. of the annual Types seminar*, March 1999.

[6] Robert Bruce Findler. PLT DrScheme: Programming environment manual. Technical Report PLT-TR2006-3-v352, PLT Scheme Inc., 2006. See also: Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme, Journal of Functional Programming, 12(2):159–182, March 2002. http://www.ccs.neu.edu/scheme/pubs/.

[7] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[8] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, page 53, September 2000.

[9] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symposium on Logic in Computer Science*, page 204. LICS, July 1999.

[10] Oleg Kiselyov. Macros that compose: Systematic macro programming. In *Generative Programming and Component Engineering (GPCE'02)*, 2002.

[11] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*, 2004.

[12] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 2–12, Tallinn, Estonia, September 2005. ACM Press.

[13] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[14] Carsten Schürmann. Recursion for higher-order encodings. In *Proceedings of Computer Science Logic (CSL 2001)*, pages 585–599, 2001.

[15] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[16] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings*, Lecture Notes in Computer Science, pages 513–527. Springer-Verlag, Berlin, 2003.

[17] Christian Urban and Christine Tasson. Nominal reasoning techniques in isabelle/hol. In *Proceedings of the 20th Conference on Automated Deduction (CADE 2005)*, volume 3632, pages 38–53, Tallinn, Estonia, July 2005. Springer Verlag.

Concurrency Oriented Programming in Termite Scheme

Guillaume Germain Marc Feeley Stefan Monnier
Université de Montréal
{germaing, feeley, monnier}@iro.umontreal.ca

Abstract

Termite Scheme is a variant of Scheme intended for distributed computing. It offers a simple and powerful concurrency model, inspired by the Erlang programming language, which is based on a message-passing model of concurrency.

Our system is well suited for building custom protocols and abstractions for distributed computation. Its open network model allows for the building of non-centralized distributed applications. The possibility of failure is reflected in the model, and ways to handle failure are available in the language. We exploit the existence of first class continuations in order to allow the expression of high-level concepts such as process migration.

We describe the Termite model and its implications, how it compares to Erlang, and describe sample applications built with Termite. We conclude with a discussion of the current implementation and its performance.

General Terms Distributed computing in Scheme

Keywords Distributed computing, Scheme, Lisp, Erlang, Continuations

1. Introduction

There is a great need for the development of widely distributed applications. These applications are found under various forms: stock exchange, databases, email, web pages, newsgroups, chat rooms, games, telephony, file swapping, etc. All distributed applications share the property of consisting of a set of processes executing concurrently on different computers and communicating in order to exchange data and coordinate their activities. The possibility of failure is an unavoidable reality in this setting due to the unreliability of networks and computer hardware.

Building a distributed application is a daunting task. It requires delicate low-level programming to connect to remote hosts, send them messages and receive messages from them, while properly catching the various possible failures. Then it requires tedious encoding and decoding of data to send them on the wire. And finally it requires designing and implementing on top of it its own application-level protocol, complete with the interactions between the high-level protocol and the low-level failures. Lots and lots of bug opportunities and security holes in perspective.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

Termite aims to make this much easier by doing all the low-level work for you and by leveraging Scheme's powerful abstraction tools to make it possible to concentrate just on the part of the design of the high-level protocol which is specific to your application.

More specifically, instead of having to repeat all this work every time, Termite offers a simple yet high-level concurrency model on which reliable distributed applications can be built. As such it provides functionality which is often called middleware. As macros abstract over syntax, closures abstract over data, and continuations abstract over control, the concurrency model of Termite aims to provide the capability of abstracting over distributed computations.

The Termite language itself, like Scheme, was kept as powerful and simple as possible (but no simpler), to provide simple orthogonal building blocks that we can then combine in powerful ways. Compared to Erlang, the main additions are two building blocks: macros and continuations, which can of course be sent in messages like any other first class object, enabling such operations as task migration and dynamic code update.

An important objective was that it should be flexible enough to allow the programmer to easily build and experiment with libraries providing higher-level distribution primitives and frameworks, so that we can share and reuse more of the design and implementation between applications. Another important objective was that the basic concurrency model should have sufficiently clean semantic properties to make it possible to write simple yet robust code on top of it. Only by attaining those two objectives can we hope to build higher layers of abstractions that are themselves clean, maintainable, and reliable.

Sections 2 and 3 present the core concepts of the Termite model, and the various aspects that are a consequence of that model. Section 4 describes the language, followed by extended examples in Sec. 5. Finally, Section 6 presents the current implementation with some performance measurements.

2. Termite's Programming Model

The foremost design philosophy of the Scheme [14] language is the definition of a small, coherent core which is as general and powerful as possible. This justifies the presence of first class closures and continuations in the language: these features are able to abstract data and control, respectively. In designing Termite, we extended this philosophy to concurrency and distribution features. The model must be simple and extensible, allowing the programmer to build his own concurrency abstractions.

Distributed computations consist of multiple concurrent programs running in usually physically separate spaces and involving data transfer through a potentially unreliable network. In order to model this reality, the concurrency model used in Termite views the computation as a set of isolated sequential processes which are uniquely identifiable across the distributed system. They communicate with each other by exchanging messages. Failure is reflected in Termite by the uncertainty associated with the transmission of

a message: there is no guarantee that a message sent will ever be delivered.

The core features of Termite's model are: isolated sequential processes, message passing, and failure.

2.1 Isolated sequential processes

Termite processes are lightweight. There could be hundreds of thousands of them in a running system. Since they are an important abstraction in the language, the programmer should not consider their creation as costly. She should use them freely to model the problems at hand.

A Termite process executes in the context of a *node*. Nodes are identified with a *node identifier* that contains information to locate a node physically and connect to it (see Sec. 3.4 for details). The procedure `spawn` creates and starts a new process on the node of the parent process.

Termite processes are identified with *process identifiers* or *pids*. *Pids* are *universally unique*. We make the distinction here between *globally unique*, which means unique at the node level, and *universally unique*, which means unique at the whole distributed network level. A *pid* is therefore a reference to a process and contains enough information to determine the node on which the process is located. It is important to note that there is no guarantee that a *pid* refers to a process that is reachable or still alive.

Termite enforces strong isolation between each of the processes: it is impossible for a process to directly access the memory space of another process. This is meant to model the reality of a physically distributed system, and has the advantage of avoiding the problems relative to sharing memory space between processes. This also avoids having to care about mutual exclusion at the language level. There is no need for mutexes and condition variables. Another consequence of that model is that there is no need for a distributed garbage collector, since there cannot be any foreign reference between two nodes's memory spaces. On the other hand, a live process might become unreachable, causing a resource leak: this part of the resource management needs to be done manually.

2.2 Sending and receiving messages

Processes interact by exchanging messages. Each process has a single mailbox in which Termite stores messages in the order in which it receives them. This helps keep the model simple since it saves us from introducing concepts like mailboxes or ports.

In Termite, a message can be any serializable first class value. It can be an atomic value such as a number or a symbol, or a compound value such as a list, record, or continuation, as long as it contains only serializable values.

The message sending operation is asynchronous. When a process sends a message, this is done without the process blocking.

The message retrieval operation is synchronous. A process attempting to retrieve a message from its mailbox will block if no message is available.

Here is an example showing the basic operations used in Termite: A process *A* spawns a new process *B*; The process *B* sends a message to *A*; The process *A* waits until it receives it.

```
(let ((me (self)))
  (spawn
    (lambda ()
      (! me "Hello, world!"))))

(?)                                  ⟹ "Hello, world!"
```

The procedure `self` returns the *pid* of the current process. The procedure `!` is the *send message* operation, while the procedure `?` is the *retrieve the next mailbox message* operation.

2.3 Failure

The unreliability of the physical, "real world" aspects of a distributed computation makes it necessary for that computation to pay close attention to the possibility of failure. A computation run on a single computer with no exterior communication generally does not have to care whether the computer crashes. This is not the case in a distributed setting, where some parts of the computation might go on even in the presence of hardware failure or if the network connection goes down. In order to model failure, sending a message in Termite is an unreliable operation. More specifically, the semantics of the language do not specify how much time a message will take to reach its destination and it may even never reach it, e.g. because of some hardware failure or excessive load somewhere along the way. Joe Armstrong has called this *send and pray semantics* [2].

Since the transmission of a message is unreliable, it is generally necessary for the application to use a protocol with *acknowledgments* to check that the destination has received the message . The burden of implementing such a protocol is left to the application because there are several ways to do it, each with an impact on the way the application is organized. If no acknowledgment is received within a certain time frame, then the application will take some action to recover from the failure. In Termite the mechanism for handling the waiting period is to have an optional timeout for the amount of time to wait for messages. This is a basic mechanism on which we can build higher level failure handling.

3. Peripheral Aspects

Some other Termite features are also notable. While they are not core features, they come naturally when considering the basic model. The most interesting of those derived features are serialization, how to deal with mutation, exception handling and the naming of computers and establishing network connections to them.

3.1 Serialization

There should be no restrictions on the type of data that can constitute a message. Therefore, it is important that the runtime system of the language supports serialization of every first class value in the language, including closures and continuations.

But this is not always possible. Some first class values in Scheme are hard to serialize meaningfully, like ports and references to physical devices. It will not be possible to serialize a closure or a continuation if it has a direct reference to one of these objects in their environment.

To avoid having references to non-serializable objects in the environment, we build *proxies* to those objects by using processes, so that the serialization of such an object will be just a pid. Therefore, Termite uses processes to represent ports (like open files) or references to physical devices (like the mouse and keyboard).

Abstracting non-serializable objects as processes has two other benefits. First, it enables the creation of interesting abstractions. For example, a click of the mouse will send a message to some "mouse listener", sending a message to the process proxying the standard output will print it, etc. Secondly, this allows us to access non-movable resources transparently through the network.

3.2 Explicit mutation

To keep the semantics clean and simplify the implementation, mutation of variables and data structures is not available. This allows the implementation of message-passing within a given computer without having to copy the content of the message.

For this reason, Termite forbids explicit mutation in the system (as with the special form `set!` and procedures `set-car!`, `vector-set!`, etc.) This is not as big a limitation as it seems at

first. It is still possible to replace or simulate mutation using processes. We just need to abstract state using messages and suspended processes. This is a reasonable approach because processes are lightweight. An example of a mutable data structure implemented using a process is given in Section 4.6.

3.3 Exception handling

A Termite exception can be any first class value. It can be *raised* by an explicit operation, or it can be the result of a software error (like division by zero or a type error).

Exceptions are dealt with by installing dynamically scoped handlers. Any exception raised during execution will invoke the handler with the exception as a parameter. The handler can either choose to manage that exceptional condition and resume execution or to raise it again. If it raises the exception again, it will invoke the nearest encapsulating handler. Otherwise, the point at which execution resumes depends on the handler: an *exception-handler* will resume execution at the point the exception was raised, whereas an *exception-catcher* will resume execution at the point that the handler was installed.

If an exception propagates to the outer scope of the process (i.e. an uncaught exception), the process dies. In order to know who to notify of such a circumstance, each process has what we call *links* to other processes. When a process dies and it is *linked* to other processes, Termite propagates the exception to those processes. Links between processes are directed. A process which has an outbound link to another process will send any uncaught exception to the other process. Note that exception propagation, like all communication, is unreliable. The implementation will make an extra effort when delivering an exception since that kind of message may be more important for the correct execution of the application.

Receiving an exception causes it to be raised in the receiving process at the moment of the next *message retrieve* operation by that process.

Links can be established in both directions between two processes. In that situation the link is said to be *bidirectional*. The direction of the link should reflect the relation between the two processes. In a supervisor-worker relation, we will use a bidirectional link since both the supervisor and the worker need to learn about the death of the other (the supervisor so it may restart the worker, the worker so it can stop executing). In a monitor-worker relation where the monitor is an exterior observer to the worker, we will use an outbound link from the worker since the death of the monitor should not affect the worker.

3.4 Connecting nodes

Termite processes execute on nodes. Nodes connect to each other when needed in order to exchange messages. The current practice in Termite is to uniquely identify nodes by binding them to an IP address and a TCP port number. Node references contain exactly that information and therefore it is possible to reach a node from the information contained in the reference. Those references are built using the `make-node` procedure.

Termite's distributed system model is said to be *open*: nodes can be added or removed from a distributed computation at any time. Just like it is possible to spawn a process on the current node, it is possible to spawn a process on a remote node by using the `remote-spawn` procedure. This is one of the key features that enable distribution.

The concept of global environment as it exists in Scheme is tied to a node. A variable referring to the global environment will resolve to the value tied to that variable on the node on which the process is currently executing.

3.5 Tags

A process may make multiple concurrent requests to another process. Also, replies to requests may come out of order (and even from a completely different process, e.g. if the request was forwarded). In those cases, it can be difficult to sort out which reply corresponds to which request. For this purpose, Termite has a universally unique reference data type called *tag*. When needed, the programmer can then uniquely mark each new request with a new *tag*, and copy the tag into the replies, to unequivocally indicate which reply corresponds to which request. Note that this can be necessary even when there is apparently only one request pending, since the process may receive a spurious delayed reply to some earlier request which had timed out.

4. The Termite Language

This section introduces the Termite language through examples. For the sake of simplicity those examples assume that messages will always be delivered (no failure) and always in the same order that they were sent.

The fundamental operations of Termite are:

`(spawn` *`fun`*`)`: create a process running *fun* and return its *pid*.

`(!` *`pid msg`*`)`: send message *msg* to process *pid*.

`(?` `[`*`timeout`* `[`*`default`*`]]``)`: fetch a message from the mailbox.

4.1 Making a "server" process

In the following code, we create a process called `pong-server`. This process will reply with the symbol `pong` to any message that is a list of the form `(`*`pid`* `ping)` where *pid* refers to the originating process. The Termite procedure `self` returns the *pid* of the current process.

```
(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (let ((msg (?)))
          (if (and (list? msg)
                   (= (length msg) 2)
                   (pid? (car msg))
                   (eq? (cadr msg) 'ping))
              (let ((from (car msg)))
                (! from 'pong)
                (loop))
              (loop)))))))

(! pong-server (list (self) 'ping))

(?)                                ⟹ pong
```

4.2 Selective message retrieval

While the `?` procedure retrieves the next available message in the process' mailbox, sometimes it can be useful to be able to choose the message to retrieve based on a certain criteria. The selective message retrieval procedure is `(??` *`pred`* `[`*`timeout`* `[`*`default`*`]])`. It retrieves the first message in the mailbox which satisfies the predicate *pred*. If none of the messages in the mailbox satisfy *pred*, then it waits until one arrives that does or until the timeout is hit.

Here is an example of the `??` procedure in use:

```
(! (self) 1)
(! (self) 2)
```

```
(! (self) 3)

(?)                                ⟹ 1
(?? odd?)                          ⟹ 3
(?)                                ⟹ 2
```

4.3 Pattern matching

The previous `pong-server` example showed that ensuring that a message is well-formed and extracting relevant information from it can be quite tedious. Since those are frequent operations, Termite offers an ML-style pattern matching facility.

Pattern matching is implemented as a special form called `recv`, conceptually built on top of the `??` procedure. It has two simultaneous roles: selective message retrieval and data destructuring. The following code implements the same functionality as the previous *pong server* but using `recv`:

```
(define better-pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from 'ping)        ; pattern to match
           (where (pid? from)) ; constraint
           (! from 'pong)))    ; action
        (loop)))))
```

The use of `recv` here only has one clause, with the pattern `(from 'ping)` and an additional side condition (also called *where clause*) `(pid? from)`. The pattern constrains the message to be a list of two elements where the first can be anything (ignoring for now the subsequent side condition) and will be bound to the variable `from`, while the second has to be the symbol `ping`. There can of course be several clauses, in which case the first message that matches one of the clauses will be processed.

4.4 Using timeouts

Timeouts are the fundamental way to deal with unreliable message delivery. The operations for receiving messages (ie. `?`, `??`) can optionally specify the maximum amount of time to wait for the reception of a message as well as a default value to return if this timeout is reached. If no timeout is specified, the operation will wait forever. If no default value is specified, the `timeout` symbol will be raised as an exception. The `recv` special form can also specify such a timeout, with an `after` clause which will be selected after no message matched any of the other clauses for the given amount of time.

```
(! some-server (list (self) 'request argument))

(? 10) ; waits for a maximum of 10 seconds
;; or, equivalently:
(recv
 (x x)
 (after 10 (raise 'timeout)))
```

4.5 Remote procedure call

The procedure `spawn` takes a thunk as parameter, creates a process which evaluates this thunk, and returns the *pid* of this newly created process. Here is an example of an RPC server to which uniquely identified requests are sent. In this case a synchronous call to the server is used:

```
(define rpc-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag ('add a b))
           (! from (list tag (+ a b)))))
        (loop)))))

(let ((tag (make-tag)))
  (! rpc-server (list (self)
                      tag
                      (list 'add 21 21)))
  (recv
    ;; note the reference to tag in
    ;; the current lexical scope
    ((,tag reply) reply)))      ⟹ 42
```

The pattern of implementing a synchronous call by creating a tag and then waiting for the corresponding reply by testing for tag equality is frequent. This pattern is abstracted by the procedure `!?`. The following call is equivalent to the last `let` expression in the previous code:

```
(!? rpc-server (list 'add 21 21))
```

Note that the procedure `!?` can take optional *timeout* and *default* arguments like the message retrieving procedures.

4.6 Mutable data structure

While Termite's native data structures are immutable, it is still possible to implement mutable data structures using processes to represent state. Here is an example of the implementation of a mutable cell:

```
(define (make-cell content)
  (spawn
    (lambda ()
      (let loop ((content content))
        (recv
          ((from tag 'ref)
           (! from (list tag content))
           (loop content))

          (('set! content)
           (loop content)))))))

(define (cell-ref cell)
  (!? cell 'ref))

(define (cell-set! cell value)
  (! cell (list 'set! value)))
```

4.7 Dealing with exceptional conditions

Explicitly signaling an exceptional condition (such as an error) is done using the `raise` procedure. Exception handling is done using one of the two procedures `with-exception-catcher` and `with-exception-handler`, which install a dynamically scoped exception handler (the first argument) for the duration of the evaluation of the body (the other arguments).

After invoking the handler on an exception, the procedure `with-exception-catcher` will resume execution at the point where the handler was installed. `with-exception-handler`, the alternative procedure, will resume execution at the point where the exception was raised. The following example illustrates this difference:

```
(list
 (with-exception-catcher
  (lambda (exception) exception)
  (lambda ()
    (raise 42) ; this will not return
    123))                          ⟹ (42)

(list
 (with-exception-handler
  (lambda (exception) exception)
  (lambda ()
    (raise 42) ; control will resume here
    123))                          ⟹ (123)
```

The procedure `spawn-link` creates a new process, just like `spawn`, but this new process is bidirectionally linked with the current process. The following example shows how an exception can propagate through a link between two processes:

```
(catch
  (lambda (exception) #t)
  (spawn (lambda () (raise 'error)))
  (? 1 'ok)
  #f)                              ⟹ #f

(catch
  (lambda (exception) #t)
  (spawn-link (lambda () (raise 'error)))
  (? 1 'ok)
  #f)                              ⟹ #t
```

4.8 Remotely spawning a process

The function to create a process on another node is `remote-spawn`. Here is an example of its use:

```
(define node (make-node "example.com" 3000))

(let ((me (self)))
  (remote-spawn node
    (lambda ()
      (! me 'boo))))               ⟹ a-pid

(?)                                ⟹ boo
```

Note that it is also possible to establish links to remote processes. The `remote-spawn-link` procedure atomically spawns and links the remote process:

```
(define node (make-node "example.com" 3000))

(catch
  (lambda (exception) exception)
  (let ((me (self)))
    (remote-spawn-link node
      (lambda ()
        (raise 'error))))
  (? 2 'ok))                       ⟹ error
```

4.9 Abstractions built using continuations

Interesting abstractions can be defined using `call/cc`. In this section we give as an example process migration, process cloning, and dynamic code update.

Process migration is the act of moving a computation from one node to another. The presence of serializable continuations in Termite makes it easy. Of the various possible forms of process migration, two are shown here. The simplest form of migration, called here `migrate-task`, is to move a process to another node, abandoning messages in its mailbox and current links behind. For that we capture the continuation of the current process, start a new process on a remote node which invokes this continuation, and then terminate the current process:

```
(define (migrate-task node)
  (call/cc
   (lambda (k)
     (remote-spawn node (lambda () (k #t)))
     (halt!))))
```

A different kind of migration (`migrate/proxy`), which might be more appropriate in some situations, will take care to leave a process behind (a *proxy*) which will forward messages sent to it to the new location. In this case, instead of stopping the original process we make it execute an endless loop which forwards to the new process every message received:

```
(define (migrate/proxy node)
  (define (proxy pid)
    (let loop ()
      (! pid (?))
      (loop)))
  (call/cc
   (lambda (k)
     (proxy
      (remote-spawn-link
       node
       (lambda () (k #t)))))))
```

Process cloning is simply creating a new process from an existing process with the same state and the same behavior. Here is an example of a process which will reply to a `clone` message with a thunk that makes any process become a "clone" of that process:

```
(define original
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'clone)
           (call/cc
            (lambda (clone)
              (! from (list tag (lambda ()
                                  (clone #t))))))))
        (loop)))))

(define clone (spawn (!? original 'clone)))
```

Updating code dynamically in a running system can be very desirable, especially with long-running computations or in high-availability environments. Here is an example of such a dynamic code update:

```
(define server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          (('update k)
           (k #t))

          ((from tag 'ping)
           (! from (list tag 'gnop))))  ; bug
        (loop)))))
```

```
(define new-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((’update k)
           (k #t))

          ((from tag ’clone)
           (call/cc
            (lambda (k)
              (! from (list tag k)))))

          ((from tag ’ping)
           (! from (list tag ’pong))))  ; fixed
        (loop)))))

(!? server ’ping)                ⟹ gnop

(! server (list ’update (!? new-server ’clone)))

(!? server ’ping)                ⟹ pong
```

Note that this allows us to build a new version of a running process, test and debug it separately and when it is ready replace the running process with the new one. Of course this necessitates cooperation from the process whose code we want to replace (it must understand the `update` message).

5. Examples

One of the goals of Termite is to be a good framework to experiment with abstractions of patterns of concurrency and distributed protocols. In this section we present three examples: first a simple load-balancing facility, then a technique to abstract concurrency in the design of a server and finally a way to transparently “robustify” a process.

5.1 Load Balancing

This first example is a simple implementation of a load-balancing facility. It is built from two components: the first is a *meter supervisor*. It is a process which supervises workers (called *meters* in this case) on each node of a cluster in order to collect load information. The second component is the work dispatcher: it receives a closure to evaluate, then dispatches that closure for evaluation to the node with the lowest current load.

Meters are very simple processes. They do nothing but send the load of the current node to their supervisor every second:

```
(define (start-meter supervisor)
  (let loop ()
    (! supervisor
       (list ’load-report
             (self)
             (local-loadavg)))
    (recv (after 1 ’ok)) ; pause for a second
    (loop)))
```

The supervisor creates a dictionary to store current load information for each meter it knows about. It listens for the update messages and replies to requests for the node in the cluster with the lowest current load and to requests for the average load of all the nodes. Here is a simplified version of the supervisor:

```
(define (meter-supervisor meter-list)
  (let loop ((meters (make-dict)))
    (recv
      ((’load-report from load)
       (loop (dict-set meters from load)))
      ((from tag ’minimum-load)
       (let ((min (find-min (dict->list meters))))
         (! from (list tag (pid-node (car min)))))
       (loop dict))
      ((from tag ’average-load)
       (! from (list tag
                     (list-average
                      (map cdr
                           (dict->list meters)))))
       (loop dict)))))

(define (minimum-load supervisor)
  (!? supervisor ’minimum-load))

(define (average-load supervisor)
  (!? supervisor ’average-load))
```

And here is how we may start such a supervisor:

```
(define (start-meter-supervisor)
  (spawn
    (lambda ()
      (let ((supervisor (self)))
        (meter-supervisor
         (map
          (lambda (node)
            (spawn
              (migrate node)
              (start-meter supervisor)))
          *node-list*))))))
```

Now that we can establish what is the current load on nodes in a cluster, we can implement load balancing. The *work dispatching server* receives a thunk, and migrates its execution to the currently least loaded node of the cluster. Here is such a server:

```
(define (start-work-dispatcher load-server)
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag (’dispatch thunk))
           (let ((min-loaded-node
                  (minimum-load load-server)))
             (spawn
               (lambda ()
                 (migrate min-loaded-node)
                 (! from (list tag (thunk))))))))
          (loop))))))

(define (dispatch dispatcher thunk)
  (!? dispatcher (list ’dispatch thunk)))
```

It is then possible to use the procedure `dispatch` to request execution of a thunk on the most lightly loaded node in a cluster.

5.2 Abstracting Concurrency

Since building distributed applications is a complex task, it is particularly beneficial to abstract common patterns of concurrency. An example of such a pattern is a server process in a client-server organization. We use Erlang’s concept of behaviors to do that: behaviors are implementations of particular patterns of concurrent interaction.

The behavior given as example in this section is derived from the *generic server* behavior. A generic server is a process that can be started, stopped and restarted, and answers RPC-like requests.

The behavior contains all the code that is necessary to handle the message sending and retrieving necessary in the implementation of a server. The behavior is only the generic framework. To create a server we need to parameterize the behavior using a *plugin* that describes the server we want to create. A plugin contains closures (often called *callbacks*) that the generic code calls when certain events occur in the server.

A plugin only contains sequential code. All the code having to deal with concurrency and passing messages is in the generic server's code. When invoking a callback, the current server state is given as an argument. The reply of the callback contains the potentially modified server code.

A generic server plugin contains four closures. The first is for server initialization, called when creating the server. The second is for procedure calls to the server: the closure dispatches on the term received in order to execute the function call. Procedure calls to the server are synchronous. The third closure is for *casts*, which are asynchronous messages sent to the server in order to do management tasks (like restarting or stopping the server). The fourth and last closure is called when terminating the server.

Here is an example of a generic server plugin implementing a key/value server:

```
(define key/value-generic-server-plugin
  (make-generic-server-plugin
   (lambda ()                           ; INIT
     (print "Key-Value server starting")
     (make-dict))

   (lambda (term from state)            ; CALL
     (match term
       ((’store key val)
        (dict-set! state key val)
        (list ’reply ’ok state))

       ((’lookup key)
        (list ’reply (dict-ref state key) state))))

   (lambda (term state)                 ; CAST
     (match term
       (’stop (list ’stop ’normal state))))

   (lambda (reason state)               ; TERMINATE
     (print "Key-Value server terminating"))))
```

It is then possible to access the functionality of the server by using the generic server interface:

```
(define (kv:start)
  (generic-server-start-link
   key/value-generic-server-plugin))

(define (kv:stop server)
  (generic-server-cast server ’stop))

(define (kv:store server key val)
  (generic-server-call server (list ’store key val)))

(define (kv:lookup server key)
  (generic-server-call server (list ’lookup key)))
```

Using such concurrency abstractions helps in building reliable software, because the software development process is less error-prone. We reduce complexity at the cost of flexibility.

5.3 Fault Tolerance

Promoting the writing of simple code is only a first step in order to allow the development of robust applications. We also need to be able to handle system failures and software errors. Supervisors are another kind of behavior in the Erlang language, but we use a slightly different implementation from Erlang's. A *supervisor* process is responsible for supervising the correct execution of a *worker* process. If there is a failure in the worker, the supervisor restarts it if necessary.

Here is an example of use of such a supervisor:

```
(define (start-pong-server)
  (let loop ()
    (recv
      ((from tag ’crash)
       (! from (list tag (/ 1 0))))
      ((from tag ’ping)
       (! from (list tag ’pong))))
    (loop)))

(define robust-pong-server
  (spawn-thunk-supervised start-pong-server))

(define (ping server)
  (!? server ’ping 1 ’timeout))

(define (crash server)
  (!? server ’crash 1 ’crashed))

(define (kill server)
  (! server ’shutdown))

(print (ping robust-pong-server))
(print (crash robust-pong-server))
(print (ping robust-pong-server))
(kill robust-pong-server)
```

This generates the following trace (note that the messages prefixed with `info:` are debugging messages from the supervisor) :

```
(info: starting up supervised process)
pong
(info: process failed)
(info: restarting...)
(info: starting up supervised process)
crashed
pong
(info: had to terminate the process)
(info: halting supervisor)
```

The call to `spawn-thunk-supervised` return the *pid* of the supervisor, but any message sent to the supervisor is sent to the worker. The supervisor is then mostly transparent: interacting processes do not necessarily know that it is there.

There is one special message which the supervisors intercepts, and that consists of the single symbol `shutdown`. Sending that message to the supervisor makes it invoke a `shutdown` procedure that requests the process to end its execution, or terminate it if it does not collaborate. In the previous trace, the "had to terminate the process" message indicates that the process did not acknowledge the request to end its execution and was forcefully terminated.

A supervisor can be parameterized to set the acceptable restart frequency tolerable for a process. A process failing more often than a certain limit is shut down. It is also possible to specify the delay that the supervisor will wait for when sending a shutdown request to the worker.

The abstraction shown in this section is useful to construct a fault-tolerant server. A more general abstraction would be able to supervise multiple processes at the same time, with a policy determining the relation between those supervised processes (should the supervisor restart them all when a single process fails or just the failed process, etc.).

5.4 Other Applications

As part of Termite's development, we implemented two non-trivial distributed applications with Termite. *Dynamite* is a framework for developing dynamic AJAX-like web user-interfaces. We used Termite processes to implement the web-server side logic, and we can manipulate user-interface components directly from the server-side (for example through the *repl*). *Schack* is an interactive multiplayer game using Dynamite for its GUI. Players and monsters move around in a virtual world, they can pick up objects, use them, etc. The rooms of the world, the players and monsters are all implemented using Termite processes which interact.

6. The Termite Implementation

The Termite system was implemented on top of the Gambit-C Scheme system [6]. Two features of Gambit-C were particularly helpful for implementing the system: lightweight threads and object serialization.

Gambit-C supports lightweight prioritized threads as specified by SRFI 18 [7] and SRFI 21 [8]. Each thread descriptor contains the thread's continuation in the same linked frame representation used by first class continuations produced by `call/cc`. Threads are suspended by capturing the current continuation and storing it in the thread descriptor. The space usage for a thread is thus dependent on the depth of its continuation and the objects it references at that particular point in the computation. The space efficiency compares well with the traditional implementation of threads which preallocates a block of memory to store the stack, especially in the context of a large number of small threads. On a 32 bit machine the total heap space occupied by a trivial suspended thread is roughly 650 bytes. A single shared heap is used by all the threads for all allocations including continuations (see [9] for details). Because the thread scheduler uses scalable data structures (red-black trees) to represent priority queues of runnable, suspended and sleeping threads, and threads take little space, it is possible to manage millions of threads on ordinary hardware. This contributes to make the Termite model practically insensitive to the number of threads involved.

Gambit-C supports serialization for an interesting subset of objects including closures and continuations but not ports, threads and foreign data. The serialization format preserves sharing, so even data with cycles can be serialized. We can freely mix interpreted code and compiled code in a given program. The Scheme interpreter, which is written in Scheme, is in fact compiled code in the Gambit-C runtime system. Interpreted code is represented with common Scheme objects (vectors, closures created by compiled code, symbols, etc.). Closures use a flat representation, i.e. a closure is a specially tagged vector containing the free variables and a pointer to the entry point in the compiled code. Continuation frames use a similar representation, i.e. a specially tagged vector containing the continuation's free variables, which include a reference to the parent continuation frame, and a pointer to the return point in the compiled code. When Scheme code is compiled with Gambit-C's *block* option, which signals that procedures defined at top-level are never redefined, entry points and return points are identified using the name of the procedure that contains them and the integer index of the control point within that procedure. Serialization of closures and continuations created by compiled code is thus possible as long as they do not refer to non-serializable objects and the block option is used. However, the Scheme program performing the deserialization must have the same compiled code, either statically linked or dynamically loaded. Because the Scheme interpreter in the Gambit-C runtime is compiled with the block option, we can always serialize closures and continuations created by interpreted code and we can deserialize them in programs using the same version of Gambit-C. The serialization format is machine independent (endianness, machine word size, instruction set, memory layout, etc.) and can thus be deserialized on any machine. Continuation serialization allows the implementation of process migration with `call/cc`.

For the first prototype of Termite we used the Gambit-C system as-is. During the development process various performance problems were identified. This prompted some changes to Gambit-C which are now integrated in the official release:

- **Mailboxes**: Each Gambit-C thread has a mailbox. Predefined procedures are available to probe the messages in the mailbox and extract messages from the mailbox. The operation to advance the probe to the next message optionally takes a timeout. This is useful for implementing Termite's time limited receive operations.
- **Thread subtyping**: There is a `define-type-of-thread` special form to define subtypes of the builtin thread type. This is useful to attach thread local information to the thread, in particular the process links.
- **Serialization**: Originally serialization used a textual format compatible with the standard datum syntax but extended to all types and with the SRFI 38 [5] notation for representing cycles. We added hash tables to greatly improve the speed of the algorithm for detecting shared data. We improved the compactness of the serialized objects by using a binary format. Finally, we parameterized the serialization and deserialization procedures (`object->u8vector` and `u8vector->object`) with an optional conversion procedure applied to each subobject visited during serialization or constructed during deserialization. This allows the program to define serialization and deserialization methods for objects such as ports and threads which would otherwise not be serializable.
- **Integration into the Gambit-C runtime**: To correctly implement tail-calls in C, Gambit-C uses computed gotos for intra-module calls but trampolines to jump from one compilation unit to another. Because the Gambit-C runtime and the user program are distributed over several modules, there is a relatively high cost for calling procedures in the runtime system from the user program. When the Termite runtime system is in a module of its own, calls to some Termite procedures must cross two module boundaries (user program to Termite runtime, and Termite runtime to Gambit-C runtime). For this reason, integrating the Termite runtime in the thread module of the Gambit-C runtime enhances execution speed (this is done simply by adding `(include "termite.scm")` at the end of the thread module).

7. Experimental Results

In order to evaluate the performance of Termite, we ran some benchmark programs using Termite version 0.9. When possible, we compared the two systems by executing the equivalent Erlang program using Erlang/OTP version R11B-0, compiled with SMP support disabled. Moreover, we also rewrote some of the benchmarks directly in Gambit-C Scheme and executed them with version 4.0 beta 18 to evaluate the overhead introduced by Termite. In all cases we compiled the code, and no optimization flags were given to the compilers. We used the compiler GCC version 4.0.2 to compile Gambit-C, and we specified the configuration option "–

enable-single-host" for the compilation. We ran all the benchmarks on a GNU/Linux machine with a 1 GHz AMD Athlon 64, 2GB RAM and a 100Mb/s Ethernet, running kernel version 2.6.10.

7.1 Basic benchmarks

Simple benchmarks were run to compare the general performance of the systems on code which does not require concurrency and distribution. The benchmarks evaluate basic features like the cost of function calls and memory allocation.

The following benchmarks were used:

- The recursive Fibonacci and Takeuchi functions, to estimate the cost of function calls and integer arithmetic,
- Naive list reversal, to strain memory allocation and garbage collection,
- Sorting a list of random integers using the *quicksort* algorithm,
- String matching using the Smith Waterman algorithm.

The results of those benchmarks are given in Figure 1. They show that Termite is generally 2 to 3.5 times faster than Erlang/OTP. The only exception is for *nrev* which is half the speed of Erlang/OTP due to the overhead of Gambit-C's interrupt polling approach.

Test	Erlang (s)	Termite (s)
fib (34)	1.83	0.50
tak (27, 18, 9)	1.00	0.46
nrev (5000)	0.29	0.53
qsort (250000)	1.40	0.56
smith (600)	0.46	0.16

Figure 1. Basic benchmarks.

7.2 Benchmarks for concurrency primitives

We wrote some benchmarks to evaluate the relative performance of Gambit-C, Termite, and Erlang for primitive concurrency operations, that is process creation and exchange of messages.

The first two benchmarks stress a single feature. The first (*spawn*) creates a large number of processes. The first process creates the second and terminates, the second creates the third and terminates, and so on. The last process created terminates the program. The time for creating a single process is reported. In the second benchmark (*send*), a process repeatedly sends a message to itself and retrieves it. The time needed for a single message send and retrieval is reported. The results are given in Figure 2. Note that neither program causes any process to block. We see that Gambit-C and Termite are roughly twice the speed of Erlang/OTP for process creation, and roughly 3 times slower than Erlang/OTP for message passing. Termite is somewhat slower than Gambit-C because of the overhead of calling the Gambit-C concurrency primitives from the Termite concurrency primitives, and because Termite processes contain extra information (list of linked processes).

Test	Erlang (μs)	Gambit (μs)	Termite (μs)
spawn	1.57	0.63	0.91
send	0.08	0.22	0.27

Figure 2. Benchmarks for concurrency primitives.

The third benchmark (*ring*) creates a ring of 250 thousand processes on a single node. Each process receives an integer and then sends this integer minus one to the next process in the ring. When the number received is 0, the process terminates its execution after sending 0 to the next process. This program is run twice with a different initial number (K). Each process will block a total of $\lceil K/250000 \rceil + 1$ times (once for $K = 0$ and 5 times for $K = 1000000$).

With $K = 0$ it is mainly the ring creation and destruction time which is measured. With $K = 1000000$, message passing and process suspension take on increased importance. The results of this benchmark are given in Figure 3. Performance is given in microseconds per process. A lower number means better performance.

K	Erlang (μs)	Gambit (μs)	Termite (μs)
0	6.64	4.56	7.84
1000000	7.32	14.36	15.48

Figure 3. Performance for ring of 250000 processes

We can see that all three systems have similar performance for process creation; Gambit-C is slightly faster than Erlang and Termite is slightly slower. The performance penalty for Termite relatively to Gambit-C is due in part to the extra information Termite processes must maintain (like a list of links) and the extra test on message sends to determine whether they are intended for a local or a remote process. Erlang shows the best performance when there is more communication between processes and process suspension.

7.3 Benchmarks for distributed applications

7.3.1 "Ping-Pong" exchanges

This benchmark measures the time necessary to send a message between two processes exchanging *ping-pong* messages. The program is run in three different situations: when the two processes are running on the same node, when the processes are running on different nodes located on the same computer and when the processes are running on different nodes located on two computers communicating across a local area network. In each situation, we vary the volume of the messages sent between the processes by using lists of small integers of various lengths. The measure of performance is the time necessary to send and receive a single message. The lower the value, the better the performance.

List length	Erlang (μs)	Gambit (μs)	Termite (μs)
0	0.20	0.67	0.75
10	0.31	0.67	0.75
20	0.42	0.67	0.74
50	0.73	0.68	0.75
100	1.15	0.66	0.74
200	1.91	0.67	0.75
500	4.40	0.67	0.75
1000	8.73	0.67	0.75

Figure 4. Local ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on the same node.

The *local ping-pong* benchmark results in Figure 4 illustrate an interesting point: when the volume of messages grows, the performance of the Erlang system diminishes, while the performance of Termite stays practically the same. This is due to the fact that the Erlang runtime uses a separate heap per process, while the Gambit-C runtime uses a shared heap approach.

List length	Erlang (μs)	Termite (μs)
0	53	145
10	52	153
20	52	167
50	54	203
100	55	286
200	62	403
500	104	993
1000	177	2364

Figure 5. Inter-node ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on two different nodes on the same computer.

The *inter-node ping-pong* benchmark exercises particularly the serialization code, and the results in Figure 5 show clearly that Erlang's serialization is significantly more efficient than Termite's. This is expected since serialization is a relatively new feature in Gambit-C that has not yet been optimized. Future work should improve this aspect.

List length	Erlang (μs)	Termite (μs)
0	501	317
10	602	337
20	123	364
50	102	437
100	126	612
200	176	939
500	471	1992
1000	698	3623

Figure 6. Remote ping-pong: Measure of time necessary to send and receive a message of variable length between two processes running on two different computers communicating through the network.

Finally, the *remote ping-pong* benchmark additionally exercises the performance of the network communication code. The results are given in Figure 6. The difference with the previous program shows that Erlang's networking code is also more efficient than Termite's by a factor of about 2.5 for large messages. This appears to be due to more optimized networking code as well as a more efficient representation on the wire, which comes back to the relative youth of the serialization code. The measurements with Erlang show an anomalous slowdown for small messages which we have not been able to explain. Our best guess is that Nagle's algorithm gets in the way, whereas Termite does not suffer from it because it explicitly disables it.

7.3.2 Process migration

We only executed this benchmark with Termite, since Erlang does not support the required functionality. This program was run in three different configurations: when the process migrates on the same node, when the process migrates between two nodes running on the same computer, and when the process migrates between two nodes running on two different computers communicating through a network. The results are given in Figure 7. Performance is given in number of microseconds necessary for the migration. A lower value means better performance.

The results show that the main cost of a migration is in the serialization and transmission of the continuation. Comparatively, capturing a continuation and spawning a new process to invoke it is almost free.

Migration	Termite (μs)
Within a node	4
Between two local nodes	560
Between two remote nodes	1000

Figure 7. Time required to migrate a process.

8. Related Work

The **Actors** model is a general model of concurrency that has been developed by Hewitt, Baker and Agha [13, 12, 1]. It specifies a concurrency model where independent actors concurrently execute code and exchange messages. Message delivery is guaranteed in the model. Termite might be considered as an "impure" actor language, because it does not adhere to the strict "everything is an actor" model since only processes are actors. It also diverges from that model by the unreliability of the message transmission operation.

Erlang [3, 2] is a distributed programming system that has had a significant influence on this work. Erlang was developed in the context of building telephony applications, which are inherently concurrent. The idea of multiple lightweight isolated processes with unreliable asynchronous message transmission and controlled error propagation has been demonstrated in the context of Erlang to be useful and efficient. Erlang is a dynamically-typed semi-functional language similar to Scheme in many regards. Those characteristics have motivated the idea of integrating Erlang's concurrency ideas to a Lisp-like language. Termite notably adds to Erlang first class continuations and macros. It also features directed links between processes, while Erlang's links are always bidirectionals.

Kali [4] is a distributed implementation of Scheme. It allows the migration of higher-order objects between computers in a distributed setting. It uses a shared-memory model and requires a distributed garbage collector. It works using a centralized model where a node is supervising the others, while Termite has a peer-to-peer model. Kali does not feature a way to deal with network failure, while that is a fundamental aspect of Termite. It implements efficient communication by keeping a cache of objects and lazily transmitting closure code, which are techniques a Termite implementation might benefit from.

The Tube [11] demonstrates a technique to build a distributed programming system on top of an existing Scheme implementation. The goal is to have a way to build a distributed programming environment without changing the underlying system. It relies on the "code as data" property of Scheme and on a custom interpreter able to save state to code represented as S-expressions in order to implement code migration. It is intended to be a minimal addition to Scheme that enables distributed programming. Unlike Termite, it neither features lightweight isolated process nor considers the problems associated with failures.

Dreme [10] is a distributed programming system intended for open distributed systems. Objects are mobile in the network. It uses a shared memory model and implements a fault-tolerant distributed garbage collector. It differs from Termite in that it sends objects to remote processes by reference unless they are explicitly migrated. Those references are resolved transparently across the network, but the cost of operations can be hidden, while in Termite costly operations are explicit. The system also features a User Interface toolkit that helps the programmer to visualize distributed computation.

9. Conclusion

Termite has shown to be an appropriate and interesting language and system to implement distributed applications. Its core model is simple yet allows for the abstraction of patterns of distributed computation.

We built the current implementation on top of the Gambit-C Scheme system. While this has the benefit of giving a lot of freedom and flexibility during the exploration phase, it would be interesting to build from scratch a system with the features described in this paper. Such a system would have to take into consideration the frequent need for serialization, try to have processes as lightweight and efficient as possible, look into optimizations at the level of what needs to be transferred between nodes, etc. Apart from the optimizations it would also benefit from an environment where a more direct user interaction with the system would be possible. We intend to take on those problems in future research while pursuing the ideas laid in this paper.

10. Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Department of Microelectronics and Information Technology, Stockholm, Sweden, December 2003.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[4] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

[5] Ray Dillinger. SRFI 38: External representation for data with shared structure. `http://srfi.schemers.org/srfi-38/srfi-38.html`.

[6] Marc Feeley. Gambit-C version 4. `http://www.iro.umontreal.ca/~gambit`.

[7] Marc Feeley. SRFI 18: Multithreading support. `http://srfi.schemers.org/srfi-18/srfi-18.html`.

[8] Marc Feeley. SRFI 21: Real-time multithreading support. `http://srfi.schemers.org/srfi-21/srfi-21.html`.

[9] Marc Feeley. A case for the unified heap approach to erlang memory management. *Proceedings of the PLI'01 Erlang Workshop*, September 2001.

[10] Matthew Fuchs. *Dreme: for Life in the Net*. PhD thesis, New York University, Computer Science Department, New York, NY, United States, July 2000.

[11] David A. Halls. *Applying mobile code to distributed systems*. PhD thesis, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom, December 1997.

[12] C. E. Hewitt and H. G. Baker. Actors and continuous functionals. In E. J. Neuhold, editor, *Formal Descriptions of Programming Concepts*. North Holland, Amsterdam, NL, 1978.

[13] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[14] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

Interaction-Safe State for the Web

Jay McCarthy Shriram Krishnamurthi
Brown University
jay@cs.brown.edu sk@cs.brown.edu

Abstract

Recent research has demonstrated that continuations provide a clean basis to describe interactive Web programs. This account, however, provides only a limited description of state, which is essential to Web applications. This state is affected by the numerous control operators (known as navigation buttons) in Web browsers, which make Web applications behave in unexpected and even erroneous ways.

We describe these subtleties as discovered in the context of working Web applications. Based on this analysis we present linguistic extensions that accurately capture state in the context of the Web, presenting a novel form of dynamic scope. We support this investigation with a formal semantics and a discussion of applications. The results of this paper have already been successfully applied to working applications.

1. Introduction

The Web has become one of the most effective media for software deployment. Users no longer need to download large run-time systems, and developers are free to use their choice of programming language(s). Web browsers have grown in sophistication, enabling the construction of interfaces that increasingly rival desktop applications. The ability to centralize data improves access and reliability. Finally, individual users no longer need to install or upgrade software, since this can be done centrally and seamlessly on the server, realizing a vision of always up-to-date software.

Set against these benefits, Web application developers must confront several problems. One of the most bothersome is the impact of the stateless Web protocol on the structure of the source program. The protocol forces developers to employ a form of continuation-passing style, where the continuation represents the computation that would otherwise be lost when the server terminates servlet execution at each interaction point. Recent research demonstrates that using continuations in the source reinstates the structure of the program [13, 14, 16, 22].

Another source of difficulty is the Web browser itself. Browsers permit users to perform actions such as cloning windows or clicking the Back button. These are effectively (extra-linguistic) control operators, because they have an effect on the program's control flow. The interaction between these and state in the program can have sufficiently unexpected consequences that it induces errors even in major commercial Web sites [18]. The use of continuations does not eliminate these problems because continuations do not close over the values of mutable state.

One solution to this latter problem would be to disallow mutation entirely. Web applications do, however, contain stateful elements—e.g., the content of a shopping cart—that must persist over the course of a browsing session. To describe these succinctly and modularly (i.e., without transforming the entire program into a particular idiomatic style), it is natural to use mutation. It is therefore essential to have mutable server-side state that accounts for user interactions.

In this paper, we present a notion of state that is appropriate for interactive Web applications. The described *cells* are mutable, and follow a peculiar scoping property: rather than being scoped over the syntactic tree of expressions, they are scoped over a dynamic tree of Web interactions. To motivate the need for this notion of state, we first illustrate the kinds of interactions that stateful code must support (Sec. 2). We then informally explain why traditional state mechanisms (that are, faultily, used in some existing Web applications) fail to demonstrate these requirements (Sec. 3) and then introduce our notion of state (Sec. 4) with a semantics (Sec. 5). We also briefly discuss applications (Sec. 6) and performance (Sec. 7) from deployed applications.

2. Motivation

The PLT Scheme Web server [14], a modern Web server implemented entirely in Scheme, is a test-bed for experimenting with continuation-based Web programming. The server runs numerous working Web applications. One of the most prominent is CONTINUE [15, 17], which manages the paper submission and review phases of academic conferences. CONTINUE has been used by several conferences including Compiler Construction, the Computer Security Foundations Workshop, the ACM Symposium on Principles of Programming Languages, the International Symposium on Software Testing and Analysis, the ACM Symposium on Software Visualization, and others.

CONTINUE employs a sortable list display component. This component is used in multiple places, such as the display of the list of the papers submitted to the conference. The component has the following behaviors: the sort strategy may be reversed (i.e., it may be shown in descending or ascending order); when the user presses the Back button after changing the sort, the user sees the list sorted as it was prior to the change; the user may clone the window and explore the list with different sorts in different browser windows, without the sort used in one window interfering with the sort used in another; and, when the user returns to the list after a detour into some other part of the application, the list remains sorted in the same way as it was prior to the detour, while its content reflects changes to the state (e.g., it includes newly-submitted papers).

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

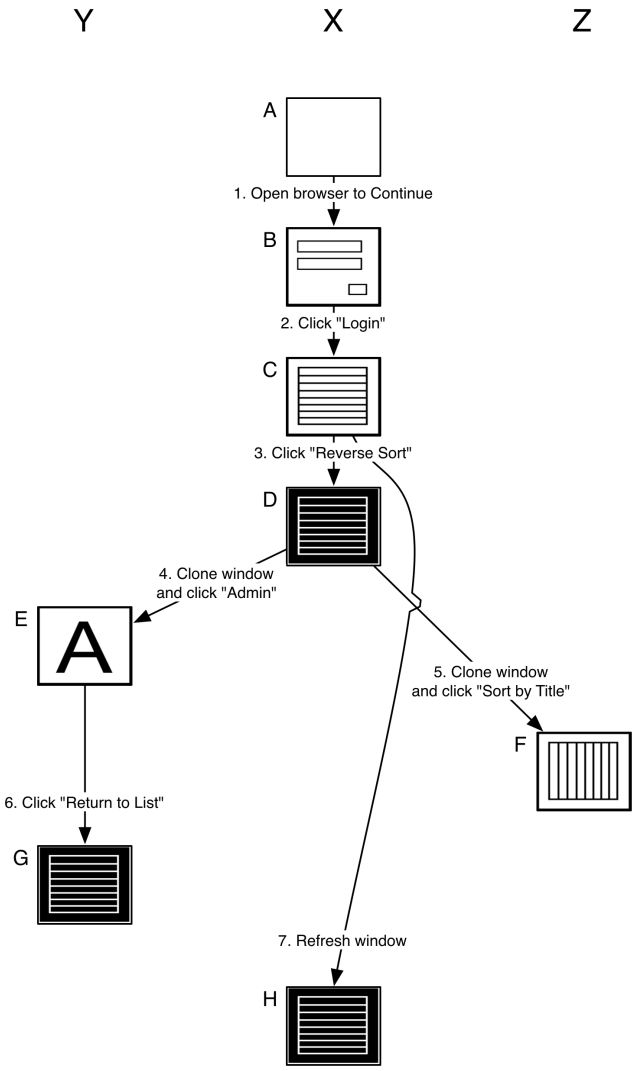


Figure 1. A diagram of an interaction with CONTINUE

Let us consider a sequence of interactions with the list of papers, and examine how we would expect this component to behave. Fig. 1 presents the outcome of these interactions. We use these interactions as our primary example throughout the paper, so it is important for the reader to study this scenario. Links in the pages that are produced by user actions are represented as links in the tree structure of the diagram. Because these actions produce multiple browser windows, the diagram has three columns, one for each browser window. Because these actions have a temporal ordering, the diagram has eight rows, one for each time step.

In this diagram, the user's browser windows are labeled X, Y, and Z. The diagram uses icons to represent the content of pages. A window with horizontal stripes represents a list sorted by author, while one with vertical stripes represents a list sorted by title. The color-inverted versions of these windows represent the same sort but in the reverse order. The 'A' icon represents the site administration page.

In Fig. 1, each node represents the page associated with a URL the user visits. A node, i, is the child of another node, j, when the page of j contains the URL of the page of i in a link or the `action` attribute of a form, and the user followed the link or submitted the form. Each edge is labeled with the action the user performed. The numbers on the edges indicate the temporal order of the actions.

When reading this diagram, it is important to recall that some user actions are not seen by the server. For example, action 4 creates a new window and then follows a link. The server is not notified of the cloning, so it only sees a request for the administrative section; presenting the generated content in window Y is the browser's responsibility.

This diagram contains some interesting interactions that highlight the requirements on the list component. First, the user clones the browser windows, which tests the facility to keep a separate and independent sort state for each browser window. This is equivalent to ensuring that the Back and Refresh buttons work correctly [20]. Second, the sort state is not lost when the user goes to a different part of the site (e.g., the Admin section in action 4) and then returns (action 6) to the list.

The placement of node H in Fig. 1 needs particular explanation. The edge leading to this node (7) is labeled with a Refresh action. Many users expect Refresh to "re-display the current page", though they implicitly expect to see updates to the underlying state (e.g., refreshing a list of email messages should display new messages received since the page was last generated). Even some browsers assume this, effectively refreshing the page when a user requests to save or print it. Under this understanding, the action 7 would simply redisplay node D.

In reality, however, the browser (if it has not cached the page) sends an HTTP request for the currently displayed URL. This request is indistinguishable from the first request on the URL, modulo timestamps, causing program execution from the previous interaction point.[1] Therefore, when the user performs action 7, the server does not receive a "redisplay" request; it instead receives a request for the content pointed to by the 'Reverse Sort' link. The server dutifully handles this request in the same way it handled the request corresponding to action 3, in this example displaying a new page that happens to look the same, modulo new papers.

Now that we have established an understanding of the desired interaction semantics of our component, we will describe the problem, introduce the solution context and then describe the solution.

3. Problem Statement and Failed Approaches

We have seen a set of requirements on the list component that have to do with the proper maintenance of state in the presence of user interactions (as shown in Fig. 1).These requirements reflect the intended state of the component, i.e., the current sort state: ordering (ascending vs. descending) and strategy (by-author vs. by-title).

We observe that values that describe the display of each page are defined by the sequence of user actions that lead to it from the root. For example, node G represents a list sorted by author in reverse ordering, because action 2 initializes the sort state to "sort by author", action 3 reverses the sort, and actions 4 and 6 do not change the sort state. To understand that the same holds true of node H, recall the true meaning of Refresh discussed earlier.

These observations indicate that there is a form of state whose modifications should be confined to the subtree rooted at the point of the modification. For example, action 4's effect is contained in the subtree rooted at node E; therefore, action 5 and node F are unaffected by action 4, because neither is in the subtree rooted at E. The challenge is to implement such state in an application.

[1] This execution can produce a drastically different outcome that would not be recognized as the "same" page, or at the very least can cause re-execution of operations that change the state: this is why, on some Web sites, printing or saving a receipt can cause billing to take place a second time.

Context

To discuss the various types of state that are available for our use, we present our work in the context of the PLT Scheme Web server. This server exposes all the underlying state mechanisms of Scheme, and should thus provide a common foundation for discussing their merits.

The PLT Scheme Web server [14] enables a direct style of Web application development that past research has found beneficial. This past research [13, 14, 16, 22] has observed that Web programs written atop the Web's CGI protocol have a form akin to continuation-passing style (CPS). A system can eliminate this burden for programmers by automatically capturing the continuation at interaction points, and resuming this captured continuation on requests.

The PLT Scheme Web server endows servlet authors with a key primitive, **send/suspend**. This primitive captures the current continuation, binds it to a URL, invokes an HTML response generation function with that URL to generate a page containing the URL, *sends* this page to the user, and then effectively *suspends* the application waiting for a user interaction via this URL. This interaction is handled by the server extracting and invoking the continuation corresponding to the URL, resuming the computation. Thus, each user interaction corresponds to the invocation of some URL, and therefore the invocation of a continuation.

We will define and implement the desired state mechanism in this context.

Failed Implementation Approaches

We first show, informally, that using the existing mechanisms of the language will *not* work. We then use this to motivate our new solution.

Studying the sequence of interactions, it is unsurprising that a purely functional approach fails to properly capture the desired semantics. Concretely, the state of the component cannot be stored as an immutable lexical binding; if it is, on return from the detour into the administrative section, the sort state reverts to the default. This is shown in Fig. 2, with the error circled in the outcome of action 7. The only alternative is to use store-passing style (SPS). Since this transformation is as invasive as CPS, a transformation that the continuation-based methodology specifically aims to avoid, we do not consider this an effective option.[2] (In Sec. 6.1 we explain why this is a practical, not ideological, concern.)

The most natural form of state is a mutable reference (called a *box* in Scheme), which is tantamount to using an entry in a database or other persistent store. This type of state fails because there is a single box for the entire interaction but, because modifications are not limited to a single subtree, different explorations can interfere with one another (a problem that is manifest in practice on numerous commercial Web sites [18]). Concretely, as Fig. 3 shows, the outcome of actions 6 and 7 would be wrong.

Since the two previous forms of state are not sensitive to continuations, it is tempting to use a form of state that is sensitive to continuations, namely **fluid-let**.[3] An identifier bound by **fluid-let** is bound for the *dynamic extent* of the evaluation of the body of the **fluid-let**. (Recall that this includes the invocation of continuations captured within this dynamic extent.)

[2] It is important to note that the administrative section implementation is just one example of where SPS would be used; SPS, like CPS, is a global transformation that would change our entire program.

[3] PLT Scheme [9] contains a feature called a *parameter*. A parameter is like a fluidly-bound identifier, except that it is also sensitive to threads. However, this distinction is not germane to our discussion, and in fact, parameters fail to satisfy us for the same reasons as **fluid-let**.

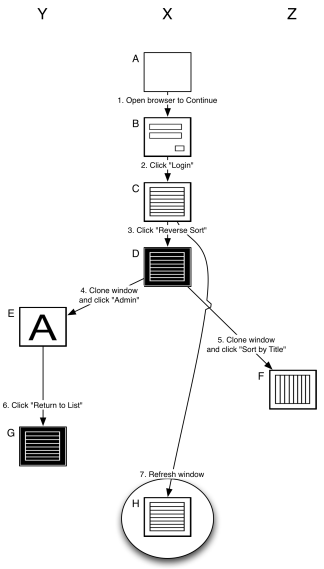


Figure 2. The interaction when *lexical bindings* are used for the sort state without SPS, where the error is circled

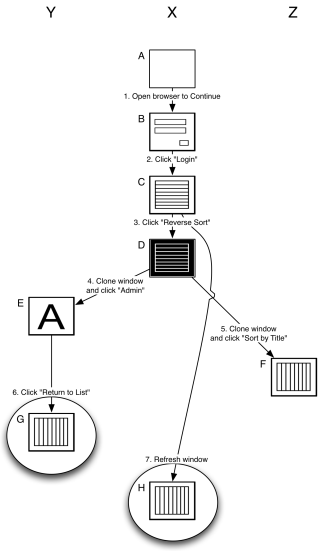


Figure 3. The interaction when a *box* is used for the sort state, where errors are circled

Fluidly-bound identifiers might seem to be a natural way of expressing state in Web applications, because they behave as if they are "closed" over continuations. Since, however, fluidly-bound identifiers are bound in a dynamic extent, any changes to their bindings are lost when the sub-computation wherein they are bound finishes. For instance, a state change in the administrative section would be lost when computation returns out of the extent of that section (here, after action 6). The only alternative is to keep that dynamic extent alive, which would require the entire computation to be written in terms of tail-calls: in other words, by conversion into continuation-passing.

4. Solution

The failed attempts above provide an intuition for why existing approaches to scope and state are insufficient. They fail because the scope in which certain state modifications are visible does not match the scope defined by the semantics of the list component.

4.1 Interaction-Safe State

We define a state mechanism called *Web cells* to meet the state management demands of the list component and similar applications.

An abstraction of Fig. 1 is used to define the semantics of cells. This abstraction is called an *interaction tree*. This tree represents the essential structure of Web interactions and the resolved values of Web cells in the program at each page, and takes into account the subtleties of Refresh, etc. The nodes of this tree are called *frame*s. Frames have one incoming edge. This edge represents the Web interaction that led to the page the frame represents.

Cells are bound in frames. The value of a cell in a frame is defined as either (a) the value of a binding for the cell in the frame; or (b) the value in the frame's parent. This definition allows cell bindings to *shadow* earlier bindings.

Evaluation of cell operations is defined relative to an evaluation context and a frame, called the current frame. The continuations captured by our Web server are designed to close over the current frame—which, in turn, closes over its parent frame, and so on up to the root—in the interaction tree. When a continuation is invoked, it reinstates its current frame (and hence the sequence of frames) so that Web cell lookup obtains the correct values.

Fig. 4 shows the interaction tree after the interactions described by Fig. 1. The actions that modify the sort state create a cell binding in the current frame. For example, when the user logs in to CONTINUE in action 2, the application stores the sort state in frame C with its default value $author$; and, during action 3, $sort$ is bound to $rev(author)$ in frame D. In action 6, the value of $sort$ is $rev(author)$, because this is the binding in frame D, which is the closest frame to G with a binding for $sort$.

The semantics of Web cells is explicitly similar to the semantics of **fluid-let**, except that we have separated the notion of evaluation context and the context of binding. Recall that with **fluid-let**, a dynamic binding is in effect for the evaluation of a specific subexpression. With Web cells, bindings affect evaluations where the current frame is a child of the binding frame.

4.2 Implementation

To implement the above informal semantics, we must describe how frames can be associated with Web interactions in a continuation-based server, so that the relation among frames models the interaction tree accurately. We describe this in the context of our canonical implementation.

Each frame, i.e., node, in the interaction tree is reached by a single action. We regard this single action to be the *creator* of the frame. When the action creates the frame, the frame's parent is the current frame of the action's evaluation. Each invocation of the continuation must create a new frame to ensure the proper behavior with regards to Refresh, as discussed in Sec. 2. Furthermore, each action corresponds to an *invocation* of a continuation. In our implementation, we must ensure that we distinguish between continuation capture and invocation. Therefore, we must change the operation that captures continuations for URLs, **send/suspend**, to create a frame when the continuation is invoked. We will describe how this is done below after we introduce the Web cell primitives.

We summarize the Web cell primitives:

- (**push-frame!**)
  Constructs an empty frame, with the current frame as its parent, and sets the new frame as the current frame.

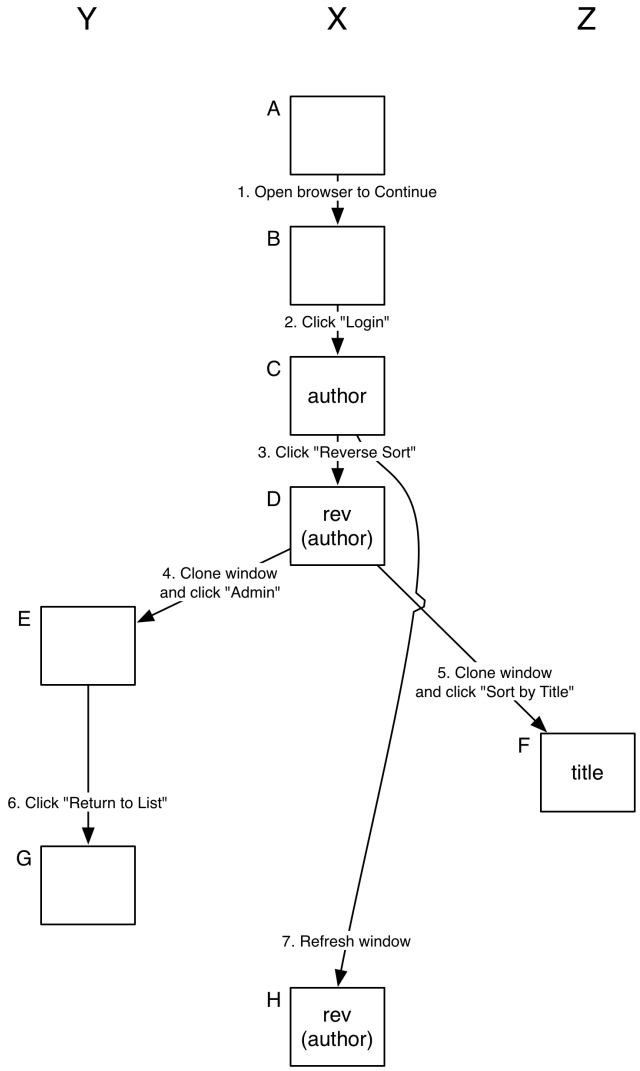


Figure 4. A diagram of an interaction with CONTINUE, labeled with cells

- (**make-cell** *initial-value*)
  Constructs and returns an opaque Web cell with some initial value, storing it in the current frame.
- (**cell-ref** *cell*)
  Yields the value associated with the cell by locating the nearest enclosing frame that has a binding for the given cell.
- (**cell-shadow** *cell new-value*)
  Creates a binding for the cell in the current frame associating the new value with the cell.

We now re-write **send/suspend** to perform the frame creation accurately. The definition is given in Fig. 5.

To show the other primitives in context, we present an example in Fig. 6. Rather than the list example, which is complicated and requires considerable domain-specific code, we present a simple counter. In this code, the boxed identifier is the interaction-safe Web cell. (The code uses the quasiquote mechanism of Scheme to

```
(define (send/suspend response-generator)
 (begin0
  (let/cc k
   (define k-url (save-continuation! k))
   (define response (response-generator k-url))
   (send response)
   (suspend))
  (push-frame!)))
```

Figure 5. A **send/suspend** that utilizes **push-frame!**

```
(define the-counter (make-cell 0))

(define (counter)
 (define request
  (send/suspend
   (λ (k-url)
    `(html
      (h2 ,(number→string (cell-ref the-counter)))
      (form ((action ,k-url))
       (input ((type "submit") (name "A") (value "Add1")))
       (input ((type "submit") (name "E") (value "Exit"))))))))
 (let ((bindings (request-bindings request)))
  (cond
   ((exists-binding? 'A bindings)
    (cell-shadow
     the-counter
     (add1 (cell-ref the-counter)))
    (counter))
   ((exists-binding? 'E bindings)
    'exit))))

(define (main-page)
 (send/suspend
  (λ (k-url)
   `(html (h2 "Main Page")
          (a ((href ,k-url))
             "View Counter"))))
  (counter)
  (main-page))
```

Figure 6. A Web counter that uses Web cells

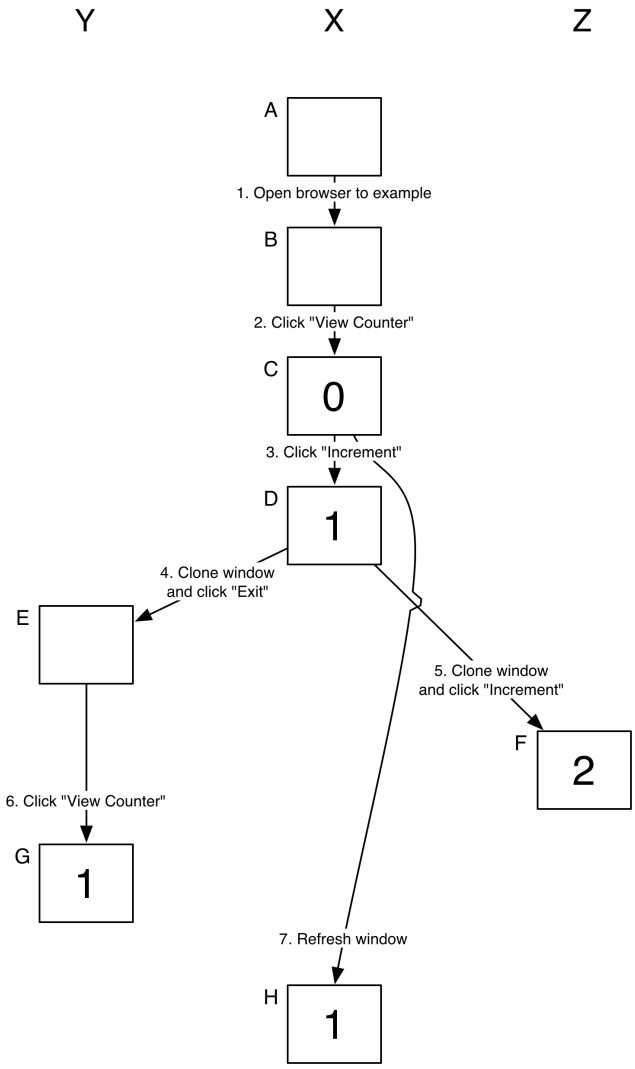


Figure 7. An interaction with the counter (Fig. 6), *structurally identical* to the CONTINUE interaction

represent HTML as an S-expression, and a library function, *exists-binding?*, to check which button the user chose.)

We present interactions with the counter application implemented by the example code in Fig. 6 through the interaction tree diagram in Fig. 7. Like Fig. 1, the links are labeled with the action the user performs. The content of each node represents the value of the counter displayed on the corresponding page. These interactions are specifically chosen to construct a tree that is structurally close to Fig. 1. Therefore, this small example shows the essence of the flow of values in the example from Sec. 2.

The next section formalizes this intuitive presentation of the Web cell primitives.

5. Semantics

The operational semantics, λ_{FS}, is defined by a context-sensitive rewriting system in the spirit of Felleisen and Hieb [7], and is a variant of the λ-calculus with **call/cc** [6] that has been enhanced with terms for representing cells and frames. Evaluation contexts are represented by the nonterminal E and allow evaluations from left-to-right in applications, including in the arguments to the built-in cell manipulation terms.

The semantics makes use of the observation that the only operations on the interaction tree are leaf-to-root lookup and leaf-append. Therefore the semantics, and eventually the implementation, only has to model the current path as a stack of frames. Lookup corresponds to walking this stack, while adding a new node corresponds to pushing a frame onto the stack.

The syntax is given in Fig. 8. The semantics makes use of the domains defined by Fig. 9 for representing stores, frames, and the frame stack. The semantics is defined by the evaluation context grammar and relations in Fig. 10 and the reduction steps in Fig. 11.

The semantics uses short-hand for representing the frame stack in the store. Each frame, ϕ, resides in the store, and $\phi[n \mapsto l]$ represents modification of the store location. The frame stack is represented by the parent pointers in each frame. When a new frame

$$\cdot\,;\ \cdot\ ;\ \cdot :: \text{Store} \times \text{Frame Stack} \times \text{Expression} \longrightarrow \text{Store} \times \text{Frame Stack} \times \text{Expression}$$

$$\mu\ ;\ \Phi\ ;\ E[((\lambda\ (x_1 \ldots x_n)\ e)\ v_1 \ldots v_n)] \longrightarrow \mu\ ;\ \Phi\ ;\ E[e[x_1/v_1, \ldots, x_n/v_n]]$$

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc}\ e)] \longrightarrow \mu\ ;\ \Phi\ ;\ E[(e\ (\lambda\ (x)\ (\textbf{abort}\ \Phi\ E[x])))]$$

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{abort}\ \Phi'\ e)] \longrightarrow \mu\ ;\ \Phi'\ ;\ e$$

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{push-frame!})] \longrightarrow \mu\ ;\ (\emptyset, \Phi)\ ;\ E[(\lambda\ (x)\ x)]$$

$$\mu\ ;\ (\phi, \Phi)\ ;\ E[(\textbf{make-cell}\ v)] \longrightarrow \mu[l \mapsto v]\ ;\ (\phi[n \mapsto l], \Phi)\ ;\ E[(\textbf{cell}\ n)]$$
where n and l are fresh

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{cell-ref}\ (\textbf{cell}\ n))] \longrightarrow \mu\ ;\ \Phi\ ;\ E[\ell(\mu, \Phi, n)]$$

$$\mu\ ;\ (\phi, \Phi)\ ;\ E[(\textbf{cell-shadow}\ (\textbf{cell}\ n)\ v)] \longrightarrow \mu[l \mapsto v]\ ;\ (\phi[n \mapsto l], \Phi)\ ;\ E[(\textbf{cell}\ n)]$$
where l is fresh

Figure 11. The reduction steps of λ_{FS}

$v ::=\ (\lambda\ (x\ \ldots)\ e)$ (*abstractions*)
$\quad\ |\ c$
$c ::=\ (\textbf{cell}\ n)$ (*cells*)
$\quad$ where n is an integer
$e ::=\ v$ (*values*)
$\quad\ |\ x$ (*identifiers*)
$\quad\ |\ (e\ e\ \ldots)$ (*applications*)
$\quad\ |\ (\textbf{call/cc}\ e)$ (*continuation captures*)
$\quad\ |\ (\textbf{abort}\ \Phi\ e)$ (*program abortion*)
$\quad$ where Φ is a frame stack (Fig. 9)
$\quad\ |\ (\textbf{push-frame!})$ (*frame creation*)
$\quad\ |\ (\textbf{make-cell}\ e)$ (*cell creation*)
$\quad\ |\ (\textbf{cell-ref}\ e\ e)$ (*cell reference*)
$\quad\ |\ (\textbf{cell-shadow}\ e\ e)$ (*cell shadowing*)

Figure 8. Syntax of λ_{FS}

Stores
$\mu ::$ Store
$\mu ::= \emptyset$ (*empty store*)
$\quad\ |\ \mu[l \mapsto v]$ (*location binding*)

Frames
$\phi ::$ Frame
$\phi ::= \emptyset$ (*empty frame*)
$\quad\ |\ \phi[x \mapsto l]$ (*cell identifier binding*)

Frame Stack
$\Phi ::$ Frame Stack
$\Phi ::= \varnothing$ (*empty frame stack*)
$\quad\ |\ \phi, \Phi$ (*frame entry*)

Figure 9. The semantic domains of λ_{FS}

is created, it is placed in the store with its parent as the old frame stack top.

The semantics is relatively simple. The cell and frame operations are quite transparent. We have included **call/cc/frame** (in Fig. 10) as an abbreviation, rather than a reduction step, to keep the semantics uncluttered. If we were to encode it as a reduction step, that step would be:

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc/frame}\ e)] \longrightarrow$$
$$\mu\ ;\ \Phi\ ;\ E[e\ (\lambda\ (x)\ (\textbf{abort}\ \Phi\ E[(seqn\ (\textbf{push-frame!})\ x)]))]$$

The order of frame creation in **call/cc/frame** is important. The implementation must ensure that each invocation of a continuation has a unique frame, and therefore a Refresh does not share the same frame as the initial request. The following *faulty* reduction step fails to ensure that each invocation has a unique frame:

$$\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc/frame}\ e)] \longrightarrow$$
$$\mu\ ;\ \Phi\ ;\ E[e\ (\lambda\ (x)\ (\textbf{abort}\ (\text{<new-frame>}, \Phi)\ E[x]))]$$

where <new-frame> is a frame constructed at capture time.

In this erroneous reduction, the new frame is created when the continuation is created, rather than each time it is invoked. Observe that the correct reduction preserves the invariant that each frame has a unique incoming edge in the interaction tree, which this reduction violates.

6. Applications

Web cells answer a pressing need of stateful Web components: they enable (a) defining stateful objects that (b) behave safely in the face of Web interactions while (c) not demanding a strong invariant of global program structure. Other techniques fail one or more of these criteria: most traditional scoping mechanisms fail (b) (as we have discussed in Sec. 4), while store-passing clearly violates (c).

Before we created Web cells, numerous PLT Scheme Web server applications—including ones written by the present authors—used to employ **fluid-let**; based on the analysis described in this paper, we have been able to demonstrate genuine errors in these applications. As a result, PLT Scheme Web server users have adopted Web cells in numerous applications, e.g., a server for managing faculty job applications, a homework turn-in application, a BibTeX front-end, a personal weblog manager, and, of course, CONTINUE itself. We present three more uses of Web cells below.

Semantics

$$eval(e) \text{ holds iff}$$
$$\emptyset\ ;\ (\emptyset, \varnothing)\ ;\ e \longrightarrow^*\ \mu\ ;\ \Phi\ ;\ v$$
$$\text{for some } \mu, \Phi, \text{ and } v$$

Evaluation Contexts

$$\begin{aligned} E\ ::=\ & [] \\ & \mid (v\ \ldots\ E\ e\ \ldots) \\ & \mid (\textbf{make-cell}\ E) \\ & \mid (\textbf{cell-ref}\ E) \\ & \mid (\textbf{cell-shadow}\ E\ e) \\ & \mid (\textbf{cell-shadow}\ v\ E) \end{aligned}$$

Cell Lookup

$$\ell :: \text{Store} \times \text{Frame Stack} \times \text{Location} \rightarrow \text{Value}$$
$$\ell(\mu, (\phi, \Phi), x) \rightarrow v \text{ iff } x \mapsto l \in \phi \text{ and } l \mapsto v \in \mu$$
$$\ell(\mu, (\phi, \Phi), x) \rightarrow \ell(\mu, \Phi, x) \text{ iff } x \mapsto l \notin \phi$$

Abbreviations

$$\begin{aligned} (let\ (x\ e_1)\ e_2) &\equiv ((\lambda\ (x)\ e_2)\ e_1) \\ (seqn\ e_1\ e_2) &\equiv (let\ (x\ c_2)\ c1) \text{ where } x \notin c_2 \\ (\textbf{call/cc/frame}\ e) &\equiv (let\ (c\ e)\ (\textbf{call/cc} \\ &\qquad (\lambda\ (v)\ (seqn\ (\textbf{push-frame!})\ (c\ v))))) \end{aligned}$$

Figure 10. The semantics of λ_{FS}

6.1 Components for Web Applications

Informally, a *component* is an abstraction that can be linked into any application that satisfies the component's published interface. Many of the tasks that Web applications perform—such as data gathering, processing, and presentation—are repetitive and stylized, and can therefore benefit from a library of reusable code.

To maximize the number of applications in which the component can be used, its interface should demand as little as possible about the enclosing context. In particular, a component that demands that the rest of the application be written in store-passing or a similar application-wide pattern is placing an onerous interface on the encapsulating application and will therefore see very little reuse. Stateful components should, therefore, encapsulate their state as much as possible.

We have built numerous Web components, including:

- `list`, whose state is the sort strategy and filter set.
- `table`, which renders a `list` component as a table split across pages, whose state is an instance of the `list` component, the number of list entries to display per page, and the currently displayed page.
- `slideshow`, whose state includes the current screen, the preferred image scale, the file format, etc.

Of the applications described above, every single one had some form of the `list` component, and a majority also had an instance of `table`—all implemented in an ad hoc and buggy manner. Many of these implementations were written using **fluid-let** and did not exhibit the correct behavior. All now use the library component instead.

6.2 Continuation Management

While Web applications should enable users to employ their browser's operations, sometimes an old continuation must expire, especially after completing a transaction. For example, once a user has been billed for the items in a shopping cart, they should not be allowed to use the Back button to change their item selection. Therefore, applications need the ability to manage their continuations.

The PLT Scheme Web server attempts to resolve this necessity by offering an operation that expires all old continuation URLs [14]. This strategy is, however, too aggressive. In the shopping cart example, for instance, only those continuations that refer to non-empty shopping carts need to be revoked: the application can be programmed to create a new shopping cart on adding the first item. In general, applications need greater control over their continuations to express fine-grained, application-specific resource management.

The interaction tree and frame stack associated with each Web continuation provide a useful mechanism to express fine-grained policies. The key feature that is missing from the existing continuation management primitives is the ability to distinguish continuations and selectively destroy them. The current frame stack of a continuation is one useful way to distinguish continuations. Thus, we extend the continuation management interface to accept a predicate on frame stacks. This predicate is used on the frame stack associated with each continuation to decide whether the continuation should be destroyed. For example, in Fig. 4 action 6's continuation could be destroyed based on the Web cell bindings of frame E, such as a hypothetical Web cell storing the identity of the logged-in user.

An application can create a predicate that identifies frames whose destruction corresponds to the above policy regarding shopping carts and purchase. First, the application must create a cell for the shopping cart session. It must then create a new session, A, when the cart goes from empty to non-empty. Then it must remove the session A when the cart becomes empty again and cause continuation destruction if the cart became empty because of a purchase. The predicate will signal destruction for continuations whose frame stack's first binding for the shopping cart session was bound to A. This particular style of continuation management enforces the policy that once a transaction has been committed, it cannot be modified via the Back button.

As another example, consider selective removal of continuations corresponding to non-idempotent requests. These requests are especially problematic in the presence of reload operations, which implicitly occur in some browsers when the user tries to save or print. We can create a cell that labels continuations and a predicate that signals the destruction of those that cannot be safely reloaded. This is a more robust solution to this problem than the Post-Redirect-Get pattern used in Web applications, as we discuss in Sec. 8.3, because it prevents the action from ever being repeated. Thus this infrastructure lets Web application developers give users maximal browsing flexibility while implementing application-specific notions of safety.

6.3 Sessions and Sub-Sessions

A session is a common Web application abstraction. It typically refers to all interactions with an application at a particular computer over a given amount of time starting at the time of login. In the Web cells framework, a session can be defined as the subtree rooted at the frame corresponding to the Logged In page. This definition naturally extends to any number of application-specific sub-session concepts. For example, in CONTINUE it is possible for the administrator to assume the identity of another user. This action

logically creates a sub-session of the session started by the initial login action.

The essential code that implements this use case is below:

```
(define-struct session (user))
(define current-session (make-cell #f))
(define (current-user)
  (session-user (cell-ref current-session)))
(define (login-as user)
  (cell-shadow current-session (make-session user)))
(define (add-review paper review-text)
  (associate-review-with-paper
   paper
   (current-user)
   review-text))
```

We now explain each step:

- When the user first logs in, the current-session cell, whose initial value is false, is shadowed by the *login-as* function.
- A new session is created and shadows the old current-session cell, when the administrator assumes the identity of another user via the *login-as* function.
- The *current-user* procedure is called whenever the current user is needed, such as by the *add-review* function. This ensures that the user is tracked by the current session, rather than any local variables.

With this strategy, an administrator can open a new window and assume the identity of a user, while continuing to use their main window for administrative actions. In doing so, the administrator need not worry about leakage of privilege through their identity, since Web cells provide a confinement of that identity in each subtree.

7. Performance

Frames and Web cells leave no significant time footprint. Their primary cost is space. The size of a frame is modest: the smallest frame consumes a mere 266 bytes (on x86 Linux). This number is dwarfed by the size of continuations, of which the smallest is twenty-five times larger. The size of the smallest frame is relevant because it represents the overhead of each frame and the cost to applications that do not use frames. CONTINUE has been used with and without frames in conferences of various sizes without noticeable performance changes in either case.

In practice, however, the space consumed depends entirely on the user's behavior and the structure of the Web application. Two factors are necessary for Web cells to adversely affect memory consumption: (1) users Refreshing URLs numerous times, and (2) the Refreshed pages not allowing further interaction (i.e., not generating additional continuations). We find that in most PLT Scheme Web server applications, most pages enable further interaction, and thus capture additional continuations. As a result, the space for continuations almost always thoroughly dominates that for frames.

8. Related Work

8.1 State and Scope

Recent research has discussed *thread-local storage* in the Java [26] and Scheme [9, 11] communities. In particular, Queinnec [22] deals with this form of scope in the context of a multi-threaded continuation-based Web server. However, in the face of continuations, thread-local store is equivalent to continuation-safe dynamic binders, i.e., parameters [9, 11]. For our purposes, these are the same as **fluid-let**, which we have argued does not solve our problem. Even so, there is much similarity between the semantics of these two types of state.

Web cells and **fluid-let** both install and modify bindings on a stack that represents a node-to-root path in a tree: Frame stacks represent paths in the interaction tree, while **fluid-let** is defined based on program stacks and the dynamic call tree. Operationally, this means that the dynamic call tree is automatically constructed for the programmer (by **fluid-let**) while frame stacks are constructed manually (by **push-frame!**), although the PLT Scheme Web server does this automatically on behalf of the programmer by burying a **push-frame!** inside the implementation of **send/suspend**. Both deal with the complication of interweaving of computation: Web continuations may be invoked any number of times and in any order, while programs with continuations may be written to have a complicated control structure with a similar property. However, to reiterate, **fluid-let** can only help us intuitively understand Web cells, as the two trees are inherently different.

Lee and Friedman [19] introduced quasi-static scope, a new form of scope that has been developed into a system for modular components, such as PLT Scheme Units [8]. This variant of scope is not applicable to our problem, as our composition is over instances of continuation invocation, rather than statically defined (but dynamically composed) software components.

First-class environments [12] are a Lisp extension where the evaluation scope of a program is explicitly controlled by the developer. This work does not allow programs to refer to their own environment in a first-class way; instead, it only allows programs to construct environments and run other programs in them. Therefore, it is not possible to express the way **make-cell** introduces a binding in whatever environment is currently active. Furthermore, this work does not define a semantics of continuations. These limitations are understandable, as first-class environments were created in the context of Symmetric Lisp as a safe way to express parallel computation. However, it may be interesting to attempt to apply our solution to a framework with environments are first-class and try to understand what extensions of such an environment are necessary to accommodate Web cells.

Olin Shivers presents BDR-scope [24] as a variant of dynamic scope defined over a finite static control-flow graph. BDR-scope differs in a fundamental way from our solution, because λ_{FS} allows a variant of dynamic scope defined over a potentially infinite dynamic control-flow tree. However, it may be possible to use Shivers's scope given an alternative representation of Web applications and an analysis that constructed the static control-flow graph representing the possible dynamic control-flows in a Web application by recognizing that recursive calls in the program represent cycles in the control-flow graph. Thus, although not directly applicable, BDR-scope may inspire future research.

Tolmach's Debugger for Standard ML [30] supports a time-travel debugging mechanism that internally uses continuations of earlier points in program executions. These continuations are captured along with the store at the earlier point in the execution. When the debugger "travels back in time", the store locations are unwound to their earlier values. Similarly, when the debugger "travels back to the future", the store is modified appropriately. The essential difference between this functionality and Web cells is that the debugger unwinds all store locations used by the program without exception, while in our context the programmer determines which values to unroll by specifying them as Web cells.

Most modern databases support nested transactions that limit the scope of effects on the database state until the transactions are committed. Therefore, code that uses a database operates with a constrained view of the database state when transactions are employed. A single Web cell representing the current transaction on the database and a database with entries for each cell models the shadowing behavior of those cells. This modeling is accomplished by creating a new transaction, A, after each new frame is created

and shadowing the `current-transaction` cell to A. Cell shadowing is possible by modifying the database state. This modification is considered shadowing, because it is only seen by transaction descended from the current transaction, i.e., frames that are descendents of the current frame. The transactions created in this modeling are never finalized. It may be interesting future work to study the meaning of and conditions for finalization and what features this implies for the Web cells model.

8.2 Web Frameworks

Other Web application frameworks provide similar features to the PLT Scheme Web server, but they often pursue other goals and therefore do not discuss or resolve the problems discussed in this paper.

Ruby on Rails [23] is a Web application framework for Ruby that provides a Model-View-Controller architecture. Rails applications are inherently defined over an Object-Relational mapping to some database. The effect of this design is that all state is shared globally or by some application-specific definition of a 'session'. Therefore, Rails cannot support the state management that Web cells offer, nor can it support many other features provided by continuations.

Seaside [5] is a Smalltalk-based Web development framework with continuation support. Seaside contains a very robust system for employing multiple components on a single page and supports a variant of Web cells by annotating object fields as being "backtrack-able." However, they do not offer a formal, or intuitive, account of the style of state we offer, and therefore do not offer comparable principles of Web application construction.

Furthermore, Seaside has other limitations relative to the PLT Scheme Web server. A single component cannot interrupt the computation to send a page to the user without passing control to another component using the `call` method, thereby precluding modal interfaces (such as alerts and prompts). The continuation URLs are not accessible to the program, inhibiting useful reusable components like an email address verifier [17]. Furthermore, Seaside's request-processing system requires a component to specify all subcomponents it might render ahead of time, decreasing the convenience of modularity.

Many Web frameworks are similar to Ruby on Rails, for example Struts [27], Mason [21] and Zope [28]; or, they pursue different goals than the PLT Scheme Web server and Seaside. MAWL [1], `<bigwig>` [2], and JWIG [4] support validation and program analysis features, such as sub-page caching and form input validation, but do not support the Back button or browser window cloning; and WASH/CGI [29] performs HTML validation, offers Back button support, and has sophisticated form field type checking and inference, but does not discuss the problems of this paper. WASH/CGI use of monadic style allows the use of store-passing style for the expression of the programs discussed in this paper. However, we have specifically tried to avoid the use of SPS, so this solution is not applicable to our context.

Java servlets [25] are an incremental improvement on CGI scripts that generally perform better. They provide a session model of Web applications and do not provide a mechanism for representing state with environment semantics, which precludes the representation of Web cells. Thus they do not offer a solution to the problem discussed in this paper.

8.3 Continuation Management

In Sec. 6.2, we discussed how Web cells can be used to organize continuation management as a solution to the problem of selectively disabling old URLs. A sub-problem of this has been addressed by work targeted at preventing the duplication of non-idempotent requests.

The Post-Redirect-Get pattern [10] is one strategy that is commonly used in many different development environments.[4] With this pattern, URLs that represent non-idempotent requests correspond to actions that generate an HTTP Redirect response, rather than an HTML page. This response redirects the browser to an idempotent URL. This strategy exploits the peculiar behavior of many browsers whereby URLs that correspond to Redirect responses are not installed in the History, and are therefore not available via the Back button. However, nothing prevents these URLs from being exposed via network dumps, network latency, or alternative browsers. In fact, this does occur in many commercial applications, forcing developers to employ a combination of HTML and JAVASCRIPT to avoid errors associated with network latency. Therefore, a continuation management strategy that can actually disable non-idempotent URLs provides a more robust, linguistic solution.

Another solution [22] to this problem relies on one-shot continuations [3]. These continuations detect when they are invoked a second time and produce a suitable error. This is easily expressed by the following abstraction:

```
(define send/suspend/once
 (λ (response-generator)
  (define called? (box #f))
  (define result (send/suspend response-generator))
  (if (unbox called?)
     (error 'send/suspend/once "Multiple invocations.")
     (begin (set-box! called? #t)
            result))))
```

However, this strategy cannot be used to implement the shopping cart example without severe transformation of the source program to propagate the *called?* binding to each code fragment that binds URLs. In contrast, our solution requires no transformations of the source, nor does it require any features of Web cells in addition to those presented.

9. Conclusion

We have demonstrated that the connection between continuations and Web computation in the presence of state is subtler than previous research suggests. In particular, a naïve approach inhibits the creation of applications with desirable interactive behavior. Our work explains the problem and provides a solution. We have implemented this solution and deployed it in several applications that are in extensive daily use.

Our result offers several directions for future work. First, we would like to construct an analysis to avoid the cost of unused frames in our implementation, similar to tail-call optimization, which avoids the cost of redundant stack frames. Second, we would like to extend our existing model checker [20] to be able to handle the subtleties introduced by this type of state management. Third, we would like to use the semantics to formally compare the expressive power of Web cells with the other primitives we have discussed in the paper. It appears that we can provide a typed account of Web cells by exploiting those for mutable references, but we have not confirmed this. Finally, we can presumably recast the result in this paper as a monad of the appropriate form.

Acknowledgments

The authors thank Matthew Flatt for his superlative work on MzScheme. Numerous people have provided invaluable feedback

[4] The name of this pattern refers to the HTTP semantics that POST requests are non-idempotent, while GET requests are idempotent. Applications such as proxy servers assume this semantics to safely cache GET requests. However, few Web applications guarantee that GET requests are, in fact, idempotent.

on this work, including Ryan Culpepper, Matthias Felleisen and Dave Herman. Several anonymous reviewers offered numerous invaluable comments on the content and presentation. We also thank the many users of the PLT Scheme Web server and the CONTINUE conference manager. This work was supported by NSF grants CCR-0305949, CPA-0429492, and CCF-0447509.

References

[1] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, 1997.

[2] C. Brabrand, A. Møller, and M. I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[3] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Language Systems*, 25(6):814–875, 2003.

[5] S. Ducasse, A. Lienhard, and L. Renggli. Seaside - a multiple control flow web application framework. In *European Smalltalk User Group - Research Track*, 2004.

[6] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[8] M. Flatt and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[9] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, Sept. 1999.

[10] A. J. Flavell. Redirect in response to POST transaction, 2000. `http://ppewww.ph.gla.ac.uk/%7Eflavell/www/post-redirect.html`.

[11] M. Gasbichler, E. Knauel, M. Sperber, and R. A. Kelsey. How to add threads to a sequential language without getting tangled up. In *Scheme Workshop*, Oct. 2003.

[12] D. Gelernter, S. Jagannathan, and T. London. Environments as first class objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–110, 1987.

[13] P. Graham. Beating the averages, Apr. 2001. `http://www.paulgraham.com/avg.html`.

[14] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, Apr. 2001.

[15] P. W. Hopkins. Enabling complex UI in Web applications with send/suspend/dispatch. In *Scheme Workshop*, 2003.

[16] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[17] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, pages 2–16, January 2003.

[18] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen. Modeling Web interactions and errors. In D. Goldin, S. Smolka, and P. Wegner, editors, *Interactive Computation: The New Paradigm*, Springer Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[19] S.-D. Lee and D. P. Friedman. Quasi-static scoping: sharing variable bindings across multiple lexical scopes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, New York, NY, USA, 1993. ACM Press.

[20] D. R. Licata and S. Krishnamurthi. Verifying interactive Web programs. In *IEEE International Symposium on Automated Software Engineering*, pages 164–173, Sept. 2004.

[21] Mason HQ. *The Mason Manual*, 2005.

[22] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, pages 23–33, 2000.

[23] Ruby on Rails. *The Ruby on Rails Documentation*, 2005.

[24] O. Shivers. The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.

[25] Sun Microsystems, Inc. *JSR154 - Java™ Servlet 2.4 Specification*, 2003.

[26] Sun Microsystems, Inc. *The Class:ThreadLocal Documentation*, 2005.

[27] The Apache Struts Project. *The Struts User's Guide*. The Apache Software Foundation, 2005.

[28] The Zope Community. *The Zope Book*, 2005.

[29] P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Symposium on the Practical Aspects of Declarative Languages*, pages 192–208, 2002.

[30] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, Apr. 1995.

Scheme for Client-Side Scripting in Mobile Web Browsing

or AJAX-Like Behavior Without Javascript

Ray Rischpater
Rocket Mobile, Inc.
ray@rocketmobile.com

Abstract

I present an implementation of Scheme embedded within a Web browser for wireless terminals. Based on a port of TinyScheme integrated with RocketBrowser, an XHTML-MP browser running on Qualcomm BREW-enabled handsets. In addition to a comparison of the resulting script capabilities, I present the changes required to bring TinyScheme to Qualcomm BREW, including adding support for BREW components as TinyScheme data types. The resulting application supports the same kinds of dynamic client-side scripted behavior as a traditional Javascript-enabled Web browser in environments too memory constrained for a Javascript implementation.

Keywords Scheme, Web, JavaScript, AJAX, mobile computing, Qualcomm BREW

1. Introduction

In the last twenty-four months, many commercial sites have deployed highly interactive Web applications leveraging the flexibility of XHTML[1], JavaScript[2], and XML[3]-based Web services. Popular sites including Google Maps[4] have both inspired Web developers and heightened consumer expectations of Web sites. This approach has significantly decreased the apparent latency of many Web applications, enhancing the user experience for all. At the same time, wide-area wireless carriers have deployed third-generation wireless networks with performance roughly equivalent to commercial broadband to the home solutions, resulting in a marked increase by consumers in the use of wireless terminals for Web access. Moreover, in an attempt to recoup network costs and increase the average revenue per subscriber (ARPU), many wireless carriers have deployed network-based programming environments such as Qualcomm BREW[5] that enables third-party developers to create and market applications that run on subscriber terminals. These applications run the gamut from entertainment (news, sports, and games) to personalization applications that permit consumers to customize the look and feel of their terminal as they purchase screen backgrounds, ring tones, and even whole handset themes that re-skin the handset's user interface.

While custom applications for these handsets can be built entirely using the native platform in C or C++, many developers have opted for hybrid native/Web-based solutions, in which interactive content is presented in a variety of ways using both native user interface controls and XHTML rendered using either the device's native Web browser or a custom part of the application capable of displaying XHTML. Rocket Mobile's RocketBrowser, an XHTML-MP[6] capable Web browser written in C for mobile handsets running Qualcomm BREW, is an example of this approach. This application provides a framework for building Web-enabled applications that run on wireless terminals, permitting application developers to introduce new tags and Web protocols that trigger native operations (compiled into the application as C functions) as well as mix the viewing of Web pages with native interfaces built using Qualcomm BREW interfaces in C. Complex applications with client-server interactions can be built quickly using RocketBrowser as a starting point, mixing traditional browser activities (say, permitting you to browse a catalog of personalization assets such as ring tones or screen backgrounds for your terminal) with activities that require support from the platform APIs (such as purchasing and installing desired media).

While flexible—developers can freely mix the browser and traditional application metaphors—this approach has some serious limitations. Notably, software behaviors must be implemented in C or C++ and compiled into the application; there is no facility for downloading additional application components. The impact of this limitation is amplified by the typical deployment scenario, in which a developer signs the application package, a third party certifies the application for quality and network interoperability, adds their signature cryptographically, and only then can the application be released to wireless operators for distribution. Once released, the application cannot be changed without passing through this entire cycle again, making offering new program features to consumers is a costly and time-consuming process. This approach also suffers the usual limitations of solutions written in C or other low-level languages, including the need to compile code for execution, the need for explicit memory management, and the lack of support for higher-order functions.

One way to ameliorate these problems is to introduce a client-side scripting solution such as JavaScript or its kin ECMAScript[7]. A scripting language such as JavaScript has the obvious advantages of rapid-high level development on the client, permitting developers to focus on the problems at hand in the content environment rather than the mechanics of extending the Web browser's implementation. Scripting also permits downloading patches or additional application components; portions of the application written as scripts interpreted on the client side can be updated in a networked application. Oddly, although this presents obvious weaknesses in the certification requirements set by most wireless operators, most carriers today permit scripted behavior in certified applications, provided that the documentation accompanying the application provided at the time of certification provides ample explanation of what behavior may be changed through script execution dur-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

ing application operation.[1] As a result, the addition of a client-side script engine can help reduce the time-to-market for new features such as user interface enhancements.

However, while JavaScript is the de facto standard for client-side scripting in Web applications on the desktop it's not without drawbacks in mobile computing. Notably, the footprint of a JavaScript implementation is actually quite large; as I discuss on page 152 in section 5, the SpiderMonkey JavaScript runtime from the Mozilla foundation is nearly twice the size of the solution I describe here on the debugging target and significantly more complex to port and maintain. While writing a simpler JavaScript implementation is possible, doing so was not desirable in the scope of our work; the time and scope of the project preferred porting an existing code base to re-implementing in whole or part an existing solution. This was true even at the expense of compatibility, because the resulting application is used by only a handful of resources internal to our firm to create end-user prototypes and applications, not the general public.

Because of these considerations—code complexity, static memory use, and time alloted for the project—when faced the need for client-side scripting in RocketBrowser to support asynchronous transactions made popular by sites using JavaScript, I decided that embedding an implementation of Scheme[8] within the application would be an appropriate choice.

This paper describes the application of Scheme to client-side scripting within a a mobile Web browser application and shows current and future applications of the technology in commercial applications. In the following section, "Preliminaries", I first present some previous work in this area that strongly influenced my approach. I describe the work required to integrate Scheme in our application, RocketBrowser, in the subsequent section "Implementation". Because there's more to shipping an application than just providing a platform, the section "Examples" shows two ways we're currently using Scheme within the resulting application at Rocket Mobile. I summarize the memory footprint of the resulting application in the section "Results", and then close the paper by describing what we've learned in the process of using Scheme for our client-side scripts in the section titled "Future Work". Finally, an appendix provides some additional detail regarding the work involved in bringing a Scheme interpreter to the Qualcomm BREW platform.

2. Preliminaries

Both client-side scripting in Web applications and Scheme applications to Web programming have a distinguished history for so young an application of computing as the Web. Understanding this history provides a crucial understanding of my motivation in selecting Scheme as a client-side language for our browser products.

2.1 Client-Side Scripts in Web Browsing

Client-side scripting in Web browser applications is not new; the initial JavaScript implementation by Brendan Eich was provided as part of Netscape 2.0 in 1995. Scripts in Web content are introduced using the `<script>` tag, which treats its content as XML CDATA, permitting scripts to consist of un-escaped character data, like so:

```
<html>
  <body>
    <script language="javascript">
     document.write('<p>Hello world.</p>')
    </script>
  </body>
</html>
```

[1] Most operator's guidelines are vague in this regard. The position of the operator is that under no circumstances may an application interfere with a wireless terminal's use as a telephone; consequently, assurances that scripted behavior cannot interfere with telephony operations often appears ample for certification.

Using a a client-side scripting language such as JavaScript or ECMAScript, content developers can write scripts that:

- Access the content of the document. JavaScript provide access to a document's contents via the model constructed by the Web browser of the document, called the Document Object Model (DOM)[9]. The DOM provides mechanisms for accessing document objects by an optional name or unique ID as well as by an object's position within the document (e.g., "the third paragraph").
- Define functions. Scripts can define functions that can be invoked by other scripts on the same page either as parts of computation or in response to user action.
- Interact with the user interface. XHTML provides attributes to many tags, including the `<body>` tag, that permit content developers to specify a script the browser should execute when a particular user action occurs. For example, developers can use the `onmouseover` attribute to trigger a script when you move the mouse cursor over the contents of a specific tag.
- Obtain content over the network. On most newer browsers, scripts can use the browser's network stack via an object such as `XMLHTTPRequest`[11] or the Microsoft ActiveX object `XMLHTTP`[12]. Using one of these interfaces a script can create a Web query, make the request over the network, and have the browser invoke a callback when the query is completed.[2]

As these points show, the flexibility of today's Web browser applications is not simply the outcome of JavaScript the language, but rather the relationship between the scripting run-time, the ability of scripts to access the DOM and user events, and the ability of scripts to obtain data from servers on the network. The union of these characteristics enables the development of asynchronous networked applications residing entirely within a set of pages being viewed by a Web browser, a strategy popularly known as Asynchronous JavaScript and XML[10] (AJAX).

2.2 Scheme and the Web

Scheme plays an important part of many Web applications, from providing scripting for servers[14] such as Apache[14] to providing servers written in Scheme providing entire Web applications, such as the PLT Web server[15] within PLT Scheme. Much has been made in both the Lisp and Scheme communities about the relationship between XML and S-expressions; recent work on SXML[16] demonstrates the advantages of working with XML within a functional paradigm. At the level of this work, those resources did not significantly affect how I went about integrating a Scheme interpreter with the RocketBrowser, but rather helped build Scheme's credibility as the choice for this problem. As I suggest later on page 153 in section 6, SXML holds great promise in using a Scheme-enabled Web browser as the starting point for building applications that use either XML Remote Procedure Call[17] (XML-RPC) or Simple Object Access Protocol[18] (SOAP) to interact with Web services.

3. Implementation

I selected the TinyScheme[19] for its size, portability, and unencumbered license agreement. This choice turned out to be a good

[2] These objects do far more than broker HTTP[13] requests for scripts. As their name suggests, they also provide XML handling capabilities, including parsing XML in the resulting response.

one; initial porting took only a few days of part-time work, and packaging both the interpreter and foreign function interfaces to the interpreter for access to handset and browser capabilities was straightforward.

TinyScheme is a mostly R^{5}RS[8] compliant interpreter that has support for basic types (integers, reals, strings, symbols, pairs, and so on) as well as vectors and property lists. It also provides a simple yet elegant foreign function interface (FFI) that lets C code create and access objects within the C runtime as well as interface with native C code. Consisting of a little over 4500 lines of C using only a handful of standard library functions, TinyScheme was an ideal choice.

Once TinyScheme was running on the wireless terminal, I integrated the TinyScheme implementation with our RocketBrowser application. This work involved adding support for the `<script>` tag as well as attributes to several other tags (such as `<body>` and `<input>`) to permit connecting user events with Scheme functions. This involved changes to the browser's event handler and rendering engine, as well as the implementation of several foreign functions that permit scripts in Scheme to access the contents of the document and request content from the browser's network and cache layers.

3.1 Bringing Scheme to the Wireless Terminal

From a programmer's standpoint, today's wireless terminals running platforms such as Qualcomm BREW are quite similar to traditional desktop and server operating systems, despite the constrained memory and processing power. There were, however, some changes to make to TinyScheme before it could run on the wireless terminal:

- Elimination of all mutable global variables to enable the application to execute without a read-write segment when built using the ARM ADS compiler for Qualcomm BREW.
- Implementation of all references to standard C library functions, typically re-implemented as wrappers around existing Qualcomm BREW functions that play the role of standard C library functions.
- Initialization of the Scheme opcode table of function pointers at run time, rather than compile time to support the relocatable code model required by Qualcomm BREW.
- Introduction of a BREW-compatible floating point library to replace the standard C floating point library provided by ARM Ltd.
- Modification of the TinyScheme FFI mechanism to pass an arbitrary pointer to an address in the C heap to permit implementation of foreign functions that required context data without further changes to the TinyScheme interpreter itself.
- Addition of the TinyScheme type `foreign_data` to permit passing references to pointers on the C heap from the FFI layer into Scheme and back again.
- Encapsulation as a Qualcomm BREW extension. TinyScheme and its foreign function interface are packaged as Qualcomm BREW extensions, stand-alone components that are referenced by other applications through the wireless terminal's module loader and application manager.
- Capping the amount of time the interpreter spends running a script to avoid a rogue script from locking up the handset.

Readers interested in understanding these changes in more detail may consult the appendix on page 154.

3.2 Integrating Scheme with the Web Browser

The RocketBrowser application is built as a monolithic C application that uses several BREW *interfaces*—structures similar to Windows Component Object Model[24] (COM) components—as well as lightweight C structures with fixed function pointers to implement a specific interface we refer to as *glyphs*. Each RocketBrowser glyph carries data about a particular part of the Web document such as its position on the drawing canvas and what to draw as well as an interface to common operations including allocation, event handling, drawing, and destruction. This is in sharp contrast to many desktop browsers, which use a DOM to represent a document's structure. These two differences: the use of C-based component interfaces and the lack of a true DOM affected the overall approach for both porting TinyScheme and integrating the interpreter with the browser.

To permit Scheme scripts to interface with the browser through the FFI, I wanted to expose a BREW interface to the browser that would allow scripts to invoke browser functions. To do this, I chose to extend TinyScheme's types to add an additional Scheme type that could encapsulate a C structure such as a BREW interface on the C heap. I added a new union type to the structure that represents a cell on the heap, and provided a means by which foreign functions could create instances of this type. Instances of the new `foreign_data` type contain not just a pointer to an address in the application's C heap, but an integer application developers can use to denote the object's type and a function pointer to a finalizer invoked by the TinyScheme runtime when the garbage collector reclaims the cell containing the reference. This new type lets developers building foreign functions pass C pointers directly to and from the Scheme world, making both the Scheme and C code that interface with the browser clearer than referring to glyphs via other mechanisms such as unique identifiers. One such object that can be passed is a BREW interface; its corresponding BREW class id (assigned by Qualcomm) provides its type identifier, and an optional destructor handles reclaiming the object when the Scheme runtime collects the cell containing the instance. Moreover the combination of a user-definable type marker and finalizer function makes the mechanism suitable for sharing a wide variety of objects with a modicum of type safety for foreign function implementers.[3]

One challenge (which I was aware of from the beginning of the project) was the lack of a true DOM within the browser; this was in fact one of the reasons why JavaScript was a less-suitable candidate for a client-side scripting engine, as it requires a fully-featured DOM for best results. As the notion of a glyph encapsulates visible content such as a string or image, there is only a vague one-to-one correspondence between glyphs and tags, let alone glyphs and objects in the DOM. As such, the glyph list maintained by the browser is a flat representation of the document suited for low memory consumption and fast drawing sorted by relative position on a document canvas, and does not provide the hierarchical view of the document content required by a true DOM. Rather than implement the entire DOM atop the browser's list of glyphs, the resulting interface supports only access to named glyphs that correspond to specific XHTML tags such as `<input>`, `<img>`, and `<div>`. To obtain a reference to a named glyph in the current document, I introduce the foreign function `document-at`, which scans the list of glyphs and returns the first glyph found with the indicated name.

In addition to being able to access a specific piece of the DOM, developers must also be able to get and mutate key properties of any named glyph: the text contents of text glyphs such as those corresponding to the `<div>` tag, and the `src` attribute of glyphs

[3] Unfortunately, as seen in the Appendix, the resulting type checking system relies on developers writing and using functions that provide type-safe casts, a mechanism scarcely better than no type checking at all in some settings.

such as image glyphs corresponding to the `<img>` tag. (As I discuss in section 6 on page 153, later extension of this work should provide access to other glyph properties as well.) I defined foreign functions to obtain and set each of these values:

- The `glyph-src` function takes a glyph and returns the URL specified in the indicated glyph's `src` attribute.
- The `glyph-value` function takes a glyph and returns the value of the glyph, either its contents for glyphs such as those corresponding to `<div>` or the user-entered value for glyphs corresponding to `<input>` or `<textarea>`.
- The `set!glyph-src` function takes a glyph and new URL and replaces the `src` attribute with the provided URL. After the interpreter finishes executing the current script, the browser will obtain the content at the new URL and re-render the page.
- The `set!glyph-value` function takes a glyph and string, and replaces the text of the indicated glyph with the new string. After the interpreter finishes executing the current script, the browser will re-render the page with the glyph's new contents.

These functions, along with `document-at`, play the role provided by the JavaScript DOM interface within our Scheme environment.

Finally, I defined the foreign function `browser-get` to support asynchronous access to Web resources from scripts on a browser page. This function takes two arguments: a string containing a URL and a function. `browser-get` asynchronously obtains the resource at the given URL and invokes the given function with the obtained resource. This provides the same functionality as JavaScript's `XMLHTTPRequest` object to perform HTTP transactions.

4. Scheme in Client Content

Implementing a client-side scripting language for RocketBrowser was more than an exercise; I intended it to provide greater flexibility for application and content developers leveraging the platform when building commercial applications. As the examples in this section demonstrates, the results are not only practical but often more concise expressions of client behavior as well.

4.1 Responding to User Events

An immediate use we had for client-side scripting was to create a page where an image would change depending on which link on the page had focus. In JavaScript, this script changes the image displayed when the mouse is over a specific link:

```
<html>
<head>
<script language="javascript">
  function change(img_name,img_src) {
    document[img_name].src=img_src;
  }
</script>
</head>
<body>
  <center>
    <img id="watcher" src="0.jpg"
     height="60" width="80"/>
  </center>
  <br/>
  <table>
    <tr>
      <td>
        <A HREF="1.html">
         onmouseover="change('watcher','1.jpg')"
         onmouseout="change('watcher','0.jpg')">
           1
        </a>
      </td>
      <td>
        <A HREF="2.html">
         onmouseover="change('watcher','2.jpg')"
         onmouseout="change('watcher','0.jpg')">
           2
        </a>
      </td>
      <td>
        <A HREF="3.html">
         onmouseover="change('watcher','3.jpg')"
         onmouseout="change('watcher','0.jpg')">
           3
        </a>
      </td>
      <td>
        <A HREF="4.html">
         onmouseover="change('watcher','4.jpg')"
         onmouseout="change('watcher','0.jpg')">
           4
        </a>
      </td>
    </tr>
  </table>
</body>
</html>
```

This code is straightforward. A simple function `change`, taking the name of an `<img>` tag and a new URL simply sets the URL of the named image to the new URL; this causes the browser to reload and redisplay the image. Then, for each of the selectable links, the XHTML invokes this function with the appropriate image when the mouse is over the link via the `onmouseover` and `onmouseout` attributes.

In the Scheme-enabled browser, I can write:

```
<html>
<head>
<script language="scheme">
  (define resourceid-offsets '(0 1 2 3 4))
  (define focus-urls
    (list->vector
      (map
        (lambda(x)
          (string-append (number->string x) ".jpg"))
        resourceid-offsets)))

  (define on-focus-change
    (glyph-set!-src (document-at "watcher")
      (vector-ref focus-urls browser-get-focus)))
</script>
</head>
<body onfocuschange="
  (on-focus-change browser-get-focusindex)">
<center>
  <img id="watcher" src="0.jpg"
    height="60" width="80"/>
</center>
<table>
<tr>
  <td><a href="1.html">1</a></td>
  <td><a href="2.html">2</a></td>
  <td><a href="3.html">3</a></td>
  <td><a href="4.html">4</a></td>
</table>
```

```
</body>
</html>
```

There are substantial differences here, although the implementation is conceptually the same. The key difference imposing a different approach is a hardware constraint that drives the browser's capabilities: most wireless terminals lack pointing devices, substituting instead a four-way navigational pad. Thus, there's no notion of mouse events; instead, the browser provides `onfocuschange` to indicate when focus has changed from one glyph to the next. A side effect of not having a pointing device is that for any page with at least one selectable item, one item will always have focus; the navigation pad moves focus between selectable items, and a separate key actually performs the selection action (analogous to the mouse button on a PC).

The XHTML opens with a brief set of definitions to establish a vector of URLs, each corresponding to a specific image to be shown when you navigate to a link on the page. This provides a way for content developers to quickly change the images shown on the page by simply updating the list at the start of the script.

The script handling the focus change, `on-focus-change`, performs the same action as its JavaScript counterpart `change`, replacing the target `<img> src` attribute with a new URL. Instead of being supplied with the new URL, however, this function takes an index to the nth selectable glyph on the page. In this case there are four, one for each link (and one corresponding to each URL in the `focus-urls` vector). The browser invokes `on-focus-change` each time the you press a directional arrow moving the focus from one link to another, as directed by the `<body>` tag's `onfocuschange` attribute.

This coupling of XHTML and Scheme replaces several kilobytes of full-screen graphic images, a custom menu control invoked by a client-specific browser tag, and the tag itself that previously provided menus in which the center region of the screen changes depending on the focused menu item. Not only is this a clear reduction in the complexity of the application's resources, but it reduces development time as our staff artist need only provide individual components of an application's main menu, not a multi-frame image consisting of screen images of each possible selected state of the application's main menu,

4.2 Asynchronous Server Interactions

As touched upon in the introduction, a key motivation for this work is the incorporation of asynchronous client-side server interaction with remote Web services. Asynchronous transactions provide users with a more interactive experience and reduce the number of key presses required when performing an action.

Consider an application providing reverse directory lookup (in which you're seeking a name associated with a known phone number). The following XHTML provides a user interface for this application:

```
<html>
<head>
<script language="javascript">
var req;

function callback() {
  div = document.getElementById("result");
  if (req.readyState == 4 &&
      req.status == 200) {
    div.innerHTML = req.responseText;
  }
  else
  {
    div.innerHTML = "network error";
  }
}

function lookup() {
   var field = document.getElementById("number");
   var url = "http://server.com/lookup.php?phone="
    + escape(field.value);

   if ( field.value.length == 10 ) {
     req=new XMLHttpRequest();
     req.open("GET", url, true);
     req.onreadystatechange = callback;
     req.send(null);
  }
}
</script>
</head>
<body>
  <form>
    <b>Phone</b>
    <input type="text"
           value="408"
   id="number"
           onkeyup="lookup();"/>
    <div id="result"/>
</body>
</html>
```

The page has two functions and handles one user event. The function `lookup` creates a query URL with the number you've entered, and if it looks like you've entered a full telephone number, it creates an `XMLHttpRequest` object to use in requesting a server-side lookup of the name for this number. This operation is asynchronous; the `XMLHttpRequest` object will invoke the function `callback` when the request is complete. The `callback` function simply replaces the contents of the named `<div>` tag with either the results of a successful transaction or an error message in the event of a network error. This process—testing the value you enter, issuing a request if it might be a valid phone number by testing its length, and updating the contents a region of the page—is all triggered any time you change the `<input>` field on the page.

In the Scheme-enabled browser, the algorithm is exactly the same but shorter:

```
<html>
<head>
<script language="scheme">
  (define service-url
    "http://server.com/lookup.php?phone=")
  (define input-glyph (document-at "entry"))
  (define result-glyph (document-at "result"))

  (define (lookup)
   (if (eqv? (string-length
     (glyph-value input-glyph)) 10)
     (browser-get
       (string-append service-url
         (glyph-value input-glyph))
       (lambda (succeeded result)
         (if succeeded
          (set!glyph-value result-glyph result)
          (set!glyph-value result-glyph "error"))))))
</script>
</head>
<body>
```

```
  <form>
    <b>Phone</b>
    <input type="text"
      value="408"
      id="entry"
      onkeyup=
        "(lookup)"/>
    <div id="result"/>
</body>
</html>
```

For clarity, this code pre-fetches references to the glyphs corresponding to the two named XHTML tags and saves those references in the `input-glyph` and `result-glyph` variables. The function `lookup` does the same work as its JavaScript counterpart, although is somewhat simpler because the underlying interface to the browser for HTTP transactions is already created and only needs an invocation via the browser foreign function `browser-get`. Like its JavaScript cousin `XMLHTTPRequest`, it operates asynchronously, applying the provided function to the result of the Web request. This browser provides this result as a list with two elements: whether the request succeeded as a boolean in the list's first element, and the text returned by the remote server as the list's second element. In the example, I pass an anonymous function that simply updates the value of the `<div>` tag on the page with the results, just as the JavaScript `callback` function does.

This example not only shows functionality previously impossible to obtain using the browser without scripting support (a developer would likely have implemented a pair of custom tags, one processing the user input and one displaying the results, and written a fair amount of code in C for this specific functionality), but demonstrates the brevity Scheme provides over JavaScript. This brevity is important not just for developers but to limit the amount of time it takes a page to download and render, as well as space used within the browser cache.

5. Results

As the examples show, the resulting port of TinyScheme meets both the objectives of the project and provides a reasonable alternative to JavaScript. Not surprisingly, its limitations are not the language itself but the degree of integration between the script runtime and the browser.

However, TinyScheme provides two key advantages: the time elapsed from the beginning of the port, and overall memory consumption within the application. As previously noted, the small size of TinyScheme made porting a trivial task (less than twenty hours of effort).

While no JavaScript port to the wireless terminal was available for comparison, one basis of comparison is the SpiderMonkey[20] JavaScript implementation within Firefox on Microsoft Windows. Compared against the TinyScheme DLL on Microsoft Windows for the Qualcomm BREW simulator, the results are nearly 2 to 1 in favor of TinyScheme for static footprint.

Implementation	Source Files	Symbols[1]	Size[2]
SpiderMonkey	36	8537	321 KB
TinyScheme	3	605	188 KB

Where:

1. The symbol count was calculated using Source Insight's (www.sourceinsight.com) project report; this count includes all C symbols (functions and variables) in the project.
2. This size indicates the size as compiled as a Windows DLL for the appropriate application target (Firefox or Qualcomm BREW Simulator) in a release configuration.

On the wireless terminal, the TinyScheme interpreter and code required to integrate it with the browser compile to about fifty kilobytes of ARM Thumb assembly; this results in an increase of approximately 50% to the browser's static footprint. This is a sizable penalty, although in practice most application binaries for Qualcomm BREW-enabled handsets are significantly larger these days; at Rocket Mobile engineers are typically concerned more with the footprint of application resources such as images and sounds rather than the code footprint.

The static footprint is a key metric for success because the Qualcomm BREW architecture does not support any memory paging mechanism. As a result, applications must be loaded from the handset's flash file system in their entirety prior to application execution. This means that the static size directly affects the run-time RAM footprint of the browser application as well. On newer handsets this is not an issue—most mid-range handsets sold in the last eighteen months have a half-megabyte or more of RAM for application use, so a fifty kilobyte increase in memory footprint when loading an application is incidental.

In addition to the memory consumed by simply loading the Scheme runtime implementation and related support code into RAM, memory is consumed by the interpreter itself. In practice, once loaded, simply starting the interpreter consumes approximately eight kilobytes of memory; our initial Scheme definitions, derived from TinyScheme's init.scm file and containing the usual definitions for things such as `when`, `unless`, and a handful of type and arithmetic operators, consumes another ten kilobytes. Thus, the startup overhead from first loading the code into memory and then initializing the runtime before doing any useful work is about seventy kilobytes of RAM. While not insignificant on older handsets, this is not a serious impediment on handsets commercially available today; in production, we can tune this file to use only the definitions likely to be used by dynamic scripts.

Run-time performance is well within acceptable bounds for user interface tasks. The user interface example shown in section 4.1 on page 150 takes on the order of sixty milliseconds of time within the Scheme interpreter on a high-end handset (the LG-9800, based on the MSM6550 chipset from Qualcomm) to execute, resulting in no apparent latency when navigating from one link to the next. The asynchronous request example shown in section 4.2 on page 151 is somewhat slower, although the time spent executing the Scheme code is dwarfed by the latency of the cellular network in completing the request.

A final measurement of success, albeit subjective, is developer satisfaction. The general consensus of those developing content for browser-based applications at Rocket Mobile is that at the outset developing applications using scripts in Scheme is no more difficult than doing the same work in JavaScript would be. Because our Web applications are built by engineers proficient in both wireless terminal and server-side work, their strong background in object-oriented and procedural methodologies tend to slow initial adoption of Scheme, and many scripts begin looking rather procedural in nature. Over time, however, contributors have been quick to move to a more functional approach to the problems they face. A key advantage helping this transition is in providing a command-line REPL built with TinyScheme and a small file of stubs that emulate the browser's foreign function interfaces. This lets new developers prototype scripts for pages without the overhead of creating XHTML files on a server, wireless terminal, or wireless terminal emulator, and encourages experimentation with the language.

6. Future Work

Commercialization of this work at Rocket Mobile is ongoing but not yet complete for two reasons. First, to streamline the development cycle, products at Rocket Mobile are typically released as a

single binary for all commercially available handsets at the time of product release; thus the binary must meet the lowest common denominator in both static and run-set size. With several million handsets on the market today having RAM capacities under a half-megabyte, the overhead posed by the interpreter and related code prohibits a widespread release. However, this is not expected to be a significant drawback for new products aimed at recently-marketed mid- and high-tier handsets, which have much greater amounts of RAM available. In the mean time, we are using the technology in a variety of prototype and pilot projects for content providers and carriers with great success.

Second, software quality is of paramount importance to wireless carriers and subscribers. While the TinyScheme implementation has had years of use in some configurations, the underlying port to Qualcomm BREW is far from proven. At present, we are testing and reviewing the implementation of the interpreter with an eye to the types of problems that can cause application failures on wireless terminals, such as ill-use of memory (dereferencing null pointers, doubly freeing heap regions or the like). This work is ongoing, and I intend to release any results of this work to the community of TinyScheme users at large.

Another area of active investigation is to provide better interfaces via the FFI to the browser's list of glyphs and individual glyph attributes. In its present form, the `glyph-src` and `glyph-value` functions and their `set!` counterparts are workable, but somewhat clumsy. Worse, as the browser exports an increasing number of glyph attributes (such as color, size, and position), the current approach will suffer from bloating due to the number of foreign functions required to access and mutate individual glyph attributes.

An obvious future direction for this work is for the implementation to include support for SXML. While the present implementation of `browser-get` does nothing with the results returned from an HTTP transaction but pass that content on as an argument to an evaluated S-expression, client applications wishing to interact with Web services via XML-RPC or SOAP would benefit from having a parsed representation of the request results. SXML provides an ideal medium for this, because it provides easy mechanisms for querying the document tree or transforming the tree[21] in various ways to provide human-readable output which can then be set as the contents of an XHTML `<div>` region. Using this approach, a front-end to a Web service can be built entirely using the browser application and client-side scripts in Scheme that collect and process user input, submit queries to the remote Web service and present results without the need for an intermediary server to transform requests and responses from XML to XHTML.

Acknowledgments

I would like to thank Shane Conder and Charles Stearns at Rocket Mobile for their initial vetting of my approach to introducing a Scheme runtime to the browser and their support throughout the project. Notably, Charles performed much of the work on RocketBrowser required to integrate the TinyScheme engine.

References

[1] Steven Pemberton, et al. editors. *XHTML 1.0: The Extensible HyperText Markup Language.* W3C Recommendation 2002/REC-xhtml1-20020801. `http://www.w3.org/TR/2002/REC-xhtml1-20020801`. 1 August 2002.

[2] JavaScript Mozilla Foundation, Mountain View, CA, USA. `http://www.mozilla.org/js/`. 2006

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau, editors. *Extensible Markup Language (XML) 1.0.* W3 Recommendation 2004/REC-xml-20040204. `http://www.w3.org/TR/2004/REC-xml-20040204`. 4 February 2004.

[4] Google. *Google Maps API.* `http://www.google.com/apis/maps/`. 2006.

[5] Qualcomm, Inc. *BREW API Reference*. `http://www.qualcomm.com/brew/`. 2005.

[6] Open Mobile Alliance. *XHTML Mobile Profile.* `http://www.openmobilealliance.org/`. 2004.

[7] ECMA International. *ECMAScript Language Specification.* `www.ecma-international.org/publications/standards/-Ecma-262.htm`. 1999.

[8] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices, 33(9):2676, September 1998.*

[9] Arnaud Le Hors, et al, editors. *Document Object Model (DOM) Level 3 Core Specification.* W3 Recommendation 2004/REC-DOM-Level-3-Core-20040407. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`. 7 April 2004.

[10] Jesse James Garrett. *Ajax: A New Approach to Web Applications.* `http://adaptivepath.com/publications/essays/archives/000385.php`. 18 February 2005.

[11] Anne van Kesteren and Dean Jackson, editors. *The XMLHttpRequest Object.* W3 Working Draft 2006/WD-XMLHttpRequest-20060405/ `http://www.w3.org/TR/XMLHttpRequest/`. 5 April 2006.

[12] Microsoft, Inc. IXMLHTTPRequest. *MSDN Library* `http://msdn.microsoft.com/library/default.asp?-url=-/library/en-us/xmlsdk/html/63409298--0516-437d-b5af-68368157eae3.asp`. 2006.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616, Hypertext Transfer Protocol—HTTP/1.1* `http://www.w3.org/Protocols/rfc2616/rfc2616.html` The Internet Society, 1999.

[14] Todd.Gillespie and Rahul Nair. mod-scheme. `http://blufox.batcave.net/mod_scheme.html`. 2005.

[15] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. *European Symposium on Programming, pages 122-136, Apr. 2001.*

[16] Oleg Kiselyov. SXML Specification. *ACM SIGPLAN Notices, v.37(6), pp. 52-58, June 2002.*

[17] Dave Winer. *XML-RPC Specification.* `http://www.xmlrpc.com/spec`. 15 Jul 1999.

[18] Don Box, et al. *Simple Object Access Protocol (SOAP) 1.1* W3C Note 2000 NOTE-SOAP-20000508. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`. 8 May 2005.

[19] Dimitrios Souflis and Jonathan S. Shapiro. TinyScheme. `http://tinyscheme.sourceforge.net/`. 2005.

[20] Mozilla Foundation. SpiderMonkey (JavaScript-C) Engine. `https://www.mozilla.org/js/spidermonkey/`. 2006.

[21] Oleg Kiselyov. XML and Scheme (a micro-talk presentation). *Workshop on Scheme and Functional Programming 2000* `http://okmij.org/ftp/Scheme/SXML-short-paper.html`. September 17, 2000

[22] Ward Willats. How to Compile BREW Applets with WinARM 4.1.0. `http://brew.wardco.com/index.html`. May 11, 2006

[23] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/` August 1, 2006.

[24] Microsoft, Inc. COM: Component Object Model Technologies `http://www.microsoft.com/com/default.mspx` 2006.

A. Porting TinyScheme to Qualcomm BREW

While software development for wireless terminals is not nearly the constrained affair it was a scant five years ago, platforms such as Qualcomm BREW still impose many restrictions on the kinds of things that can be done in C and C++. While this does not significantly impair the development of all-new code for a mobile platform, it can make porting applications from a more traditional computing environment somewhat challenging.

In the case of the TinyScheme port, the single largest impediment was the lack of a read-write segment on the Qualcomm BREW platform.[4] Without a read-write segment, the target compiler (ARM's ADS 1.2) cannot compile applications or libraries with global variables. This makes using the standard C library impossible. Instead, applications must use a limited set of helper functions provided by Qualcomm to perform operations typically provided by the C standard and math libraries. Thus, references to C functions such as `strcmp` must reconciled with Qualcomm BREW helpers such as `STRCMP`. Rather than making numerous changes to the TinyScheme implementation (which would make merging changes from later releases difficult), I implemented a small porting layer with functions for each of the required C standard library functions. In many cases, this work was as simple as writing functions such as:

```
static __inline int
strcmp(const char *a, const char *b) {
  return STRCMP(a, b);
}
```

where `STRCMP` is the Qualcomm BREW API providing the C standard library `strcmp` facility. In addition, in a few cases such as the interpreter's use of the standard C file I/O library I had to implement the interfaces atop Qualcomm BREW's native file management functions. The resulting porting layer is approximately 300 lines of code (including comments) and can be reused when porting other code requiring the standard library to Qualcomm BREW.

Dealing with floating-point numbers without a traditional math library involved similar work; in addition to providing porting functions for numeric functions such as trigonometric operations, I also needed to deal with places where the interpreter used arithmetic operators on floating-point numbers to keep the tool chain from attempting to include the default floating-point library. When writing new code, Qualcomm suggests that developers use their helper functions for arithmetic; these provide functions for addition, subtraction, multiplication, and division. Unwilling to make such drastic changes to the TinyScheme implementation, I chose a second route. To facilitate porting, Qualcomm has made available a replacement floating-point library for basic arithmetic that does not use a read-write segment; including this library incurs an additional static use of eight kilobytes of memory. If needed, I can back this out and rewrite the functions that use floating-point arithmetic to use the Qualcomm BREW helper functions to reduce the overall footprint of the interpreter.

Along the same vein, the TinyScheme interpreter had a few global variables that I had to move to the interpreter's context `struct scheme`; these were for things like the definition of zero and one as native numbers in the interpreter's representation of integer and floating-point numbers. Similar work was required for the array of type tests and a few other global variables. More problematic, however, was the table of opcodes, defined at compile time using a series of preprocessor macros to initialize a large global array of opcodes. Although this table is constant (and can thus be loaded into the read-only segment), it results in compilation errors for another reason: Qualcomm BREW requires relocatable code, and the tool chain doesn't know what to do with function pointer references in constant variables. By moving the opcode dispatch table into the interpreter's context `struct scheme`, and adding the following snippet to the interpreter's initialization function `scheme_init`:

```
#define _OP_DEF(A,B,C,D,E,OP) \
  sc->dispatch_table[j].func = A; \
  sc->dispatch_table[j].name = B; \
  sc->dispatch_table[j].min_arity = C; \
  sc->dispatch_table[j].max_arity = D; \
  sc->dispatch_table[j].arg_tests_encoding = E; \
  j++;
{
  int j = 0;
  #include "opdefines.h"
  #undef _OP_DEF
};
```

As each Scheme opcode is defined in the opdefines.h header using the `_OP_DEF` macro, this yielded an easy solution with a minimum of changes to the existing implementation.

With this work complete, the interpreter itself was able to compile and execute on Qualcomm BREW-enabled handsets, although its packaging was less than ideal. The Qualcomm BREW platform is built around a component-oriented model similar in many ways to the original Microsoft Windows Component Object Model; packaging the Scheme interpreter as a module wrapped in a BREW interface would provide greater opportunity for reuse throughout my employer's software development efforts. The resulting module, called an extension in Qualcomm parlance, actually offers three interfaces:

- The `ISchemeInterpreter` interface exports the basic interface to the interpreter permitting developers to create an instance of the interpreter, set an output port and have it evaluate S-expressions.
- The `ISchemeFFI` interface exports the foreign function interface provided by TinyScheme in its `struct scheme_interface` structure, permitting developers familiar with Qualcomm BREW an easy way to implement foreign functions for the interpreter without needing to see or access the implementation of the interpreter itself.
- The `ISchemeFFP` interface is an abstract interface that developers implement when creating foreign functions for use with the interpreter. This interface uses the methods provided by the `ISchemeFFI` interface of a running interpreter instance to implement foreign functions. The foreign functions provided by RocketBrowser are implemented as a BREW extension implementing the `ISchemeFFP` interface.

To facilitate the `ISchemeFFI` and `ISchemeFFP` interfaces, I extended TinyScheme to support references to C heap objects as opaque Scheme cell contents through a new TinyScheme type, `foreign_data`. A cell containing a reference to a C heap object looks like this:

```
typedef struct foreign_data {
  void *p;
  uint32 clsid;
} foreign_data;

// and inside of the cell structure union:
```

[4] Tools such as WinARM[22], based on GCC[23], have recently become available that will generate "fix-up" code that creates read-write variables on the heap at runtime, although they generate additional code. In addition, Qualcomm has announced support for a toolchain for use with ARM's ADS 1.2 that does not have this limitation, but this tool was not available as I performed this work.

```
  struct {
    foreign_data *data;
    foreign_func cleanup;
  } _fd;
```

Thus, the `foreign_data` structure contains a pointer to the C heap region it references, and an unsigned double word that can contain type information, such as the native class identifier of the object being referenced. Within the cell of a foreign data object is also a pointer to a foreign function the garbage collector invokes when freeing the cell. This allows foreign functions to create instances of C heap objects for use with other foreign functions without the need for explicit creation and destruction by the application developer. This new type is supported in the same manner as other TinyScheme types, with functions available for creating instances of this type as well as testing a cell to see if it contains an instance of this type. The `display` primitive is also extended to display these objects in a manner similar to the display of foreign functions.

The type data that the `foreign_data` type carries permits developers to provide type-safe cast functions when accessing C data in foreign function interfaces. For example:

```
// Return the class of the foreign_data
static __inline AEECLSID
ISCHEMEFFITYPE_GetClass(foreign_data *p) {
  return p ? p->clsid : 0;
}

// return whether this foreign_data is
// of the desired class
static __inline boolean
ISCHEMEFFITYPE_IsInstanceOf(foreign_data *p,
                            AEECLSID cls) {
  return (boolean)(p && p->clsid == cls);
}

// Return a typeless pointer to the data
// contained by a foreign_data
static __inline void *
ISCHEMEFFITYPE_GetData(foreign_data *p) {
  return p ? p->p : NULL;
}

static __inline IShell *
ISCHEMEFFPTYPE_GetShell(foreign_data *p) {
  return
   ISCHEMEFFITYPE_IsInstanceOf(p, AEECLSID_SHELL) ?
   (IShell *)ISCHEMEFFITYPE_GetData(p) : NULL;
}
```

Using inline functions such as `ISCHEMEFFITYPE_GetShell` to ensure type-safe access to `foreign_data` wrapped data at the C layer is unfortunately a manual approach. Because developers can at any time circumvent this type-safety by accessing the contents of a `foreign_data` item directly, it must be enforced by convention and inspection.

The `foreign_data` type is also used with foreign functions in this implementation. When defining foreign functions with the interpreter, the developer can also register a `foreign_data` object and its associated destructor. This lets foreign functions keep state without needing to directly modify the TinyScheme context `struct scheme`.

Wireless terminals typically provide a watchdog timer, so that a rogue application cannot lock the handset indefinitely and prevent its use as a telephone. If application execution continues until the watchdog timer fires (typically two seconds), the handset reboots, terminating the rogue application. To avoid erroneous script errors from triggering this timer and resetting the handset, I add a similar timing mechanism to `Eval_Cycle`, as well as a function `scheme_set_eval_max` to let TinyScheme users set the watchdog timer's maximum value. If a script runs for longer than the maximum permitted time, execution is aborted and the runtime user notified with an error indicating that the script could not be completed and the runtime's state is indeterminate. The RocketBrowser application sets the maximum script execution time at 500 ms, giving ample time for simple UI operations and a more-than-adequate ceiling for remaining browser operations during page layout and drawing.

With all of this in place, extensions to the TinyScheme interpreter could be written as stand-alone BREW extensions implementing the `ISchemeFFP` interface, making dynamic loading of extensions to the TinyScheme runtime possible.

Component Deployment with PLaneT

You Want it *Where?*

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Abstract

For the past two years we have been developing PLaneT, a package manager built into PLT Scheme's module system that simplifies program development by doing away with the distinction between installed and uninstalled packages. In this paper we explain how PLaneT works and the rationales behind our major design choices, focusing particularly on our decision to integrate PLaneT into PLT Scheme and the consequences that decision had for PLaneT's design. We also report our experience as PLaneT users and developers and describe what have emerged as PLaneT's biggest advantages and drawbacks.

1. Introduction

No matter how great your programming language is, it is always harder to write a program in it yourself than it is to let someone else write it for you. That is one reason why libraries are a big deal in programming languages, big enough that some languages are associated as much with their libraries as they are with their core features (*e.g.*, Java with `java.util` and C++ with the STL).

But when you move away from the standard libraries that come built in to your programming language, you can end up paying a high price for your convenience. If your program depends on any libraries, you have to make sure those dependences are installed wherever your program will run. And any dependences those libraries have need to be installed too, and any dependences *those* libraries have, and so on. If you don't have any help with this task, it is often easier to avoid using too many libraries than it is to explain how to install dozens of dependences in a `README` file. A Scheme programmer in this situation might reasonably bemoan the fact that the choice needs to be made: why, in a language like Scheme with so many great facilities for code reuse, is it so impractical to actually reuse code?

In this paper we describe our attempt at addressing that problem by presenting the PLaneT package distribution system built into PLT Scheme [5], a component-deployment tool that aims to make package distribution and installation entirely transparent. To the extent possible, PLaneT does away with the notion of an uninstalled package: a developer can use any PLaneT package in a program just by referring to it as though it were already installed, and whenever anyone runs that program, the PLaneT system will automatically install it if necessary. Effectively, programmers get to treat *every* library, even those they write themselves, as a standard library, giving them the advantages of libraries without the pain of library deployment.

Since PLaneT retrieves packages from a centralized repository, it also establishes a component marketplace for developers; it serves as a one-stop shop where they can publish their own programs and discover useful programs written by others. In effect, PLaneT is an infrastructure that transforms PLT Scheme's effective standard library from a monolithic cathedral development effort into a bazaar where anyone can contribute and the most useful libraries rise to the top, buoyed by their quality and usefulness rather than an official blessing [11]. This bazaar-like development reinforces itself to the benefit of all PLT Scheme users: in the two years since PLaneT has been included in PLT Scheme, users have contributed nearly 300,000 lines of new Scheme code in 115 packages.

In this paper, we explain our experience with building and maintaining PLaneT, with particular attention to PLaneT's more unusual design features and how they have worked in practice. Section 2 sets the stage by briefly introducing some existing component systems such as the Comprehensive Perl Archive Network (CPAN) with particular attention to their deployment strategies. Section 3 then uses those other systems to motivate our goals for PLaneT. The next few sections explain how we tried to achieve those goals: section 4 explains our server design, section 5 explains our client design with particular attention to our choice of integrating component deployment into the language rather than keeping it as an external program, section 6 discusses how PLaneT maintains PLT Scheme tool support, and section 7 explains a few complications we foresaw and tried to address in our design. In section 8 we try to assess how well our design has stood up over the two years PLaneT has been in use.

2. Some popular component systems

Before we explain the decisions we made for PLaneT, we need to explain some background on existing component systems, how they work, and some of their pitfalls. The software development community has learned a lot about both the benefits of software components and the problems they can cause. Probably the most famous such problem is so-called "DLL hell," a term coined to describe what can happen to a Microsoft Windows installation if DLLs (for "dynamically-linked libraries," a component technology) are installed incorrectly. Under the DLL system, when a program wants some functionality from a DLL, it refers to the DLL by name only. This can become a problem because the DLL may evolve over the course of new releases; if two different programs both rely on different versions of the same library, then they cannot both be installed on the same system at the same time. Furthermore, since DLLs have no particular package deployment strategy, soft-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

ware installers have to install any necessary libraries themselves; and if an installer overwrites a DLL it can inadvertently break other programs, possibly even leading to a situation where the user's entire system is unusable. Microsoft has addressed this problem in several ways; the most recent of which is its .NET assembly format, which includes metadata that includes versioning information [10].

Neither of those systems deal with component distribution — it is the programmer's responsibility to figure out some other way to locate useful components and to arrange for them to be installed on users' computers, and if a user wants to upgrade a component then it is up to that user to find and install it. Other component systems have done work to help with these tasks, though. For instance, CPAN [1] is a central repository of Perl libraries and scripts that users can download, and tools allow users to install and upgrade these libraries automatically in some cases (more on this later). The Ruby programming language features a similar system called RubyGems [12]. Many Unix-like operating systems also feature package distribution systems that address these problems; Debian's `dpkg` system [14], for instance, provides a central repository of programs and shared libraries that users can automatically fetch, install, and upgrade upon request.

Since these systems try to do more than DLLs do, they have to solve more potential problems. The most major of these is that if a tool installs a new component that relies on functionality from other components, the tool must also install appropriate versions of those prerequisites if the user does not already have them. Identifying the entire set of uninstalled prerequisites is not too difficult given the right metadata, but automatically deciding what "appropriate" means in this context is more of a challenge. For a human, the most appropriate version of a prerequisite would probably be the one that provides all the functionality that the requiring component needs in the highest-quality way and has the fewest bugs. But an automatic tool cannot hope to figure out which package that is, so it needs to simplify the problem somehow.

The solution `CPAN.pm` (an automated client for CPAN) uses is to assume that the version of a given package with the highest version number is the most appropriate, based on the reasoning that the highest version number represents the most recent package and therefore the one with the most features and fewest bugs. That is, if the user requests package `foo`, then `CPAN.pm` finds the most recent version of `foo` it can. Furthermore, if `foo` depends on package `bar`, then `CPAN.pm` downloads the highest-numbered version of `bar` available (unless `foo` explicitly asks for a particular version or a version number in a particular numeric range).

This policy is extremely optimistic, in that it assumes all programs can use any version of a package in place of its predecessors. If for instance `bar` removes a function from its public interface in a particular release, then unless `foo` compensates for that change by releasing a new package with updated code or dependence information, it will fail to install properly. In practice this problem does not come up much, probably because most packages on CPAN only release a few different versions and those that are intended for use as libraries try to maintain backwards-compatibility. However, it can and has come up in the past, and there is no automatic way to cope with it.

Debian's `dpkg` system uses a similar underlying strategy to `CPAN.pm`'s, but has evolved a convention that serves as a coping mechanism: many Debian packages include a number directly in their names (for instance, `libc5` versus `libc6`); if a package changes and breaks backwards-compatibility, the number in the package's name changes. This way, humans looking through the package list can select the most recent version of a package for new projects without worrying that future revisions will break backwards compatibility.

The RubyGems system takes the convention a step farther; their "Rational Versioning Policy," while not technically required of packages, is strongly recommended and explicitly supported by their automatic installation tools. The rational versioning policy requires that a package's version should be a dotted triple of numbers (*e.g.*, 1.4.2). Incrementing the first number indicates a backwards-incompatible change, incrementing the second number indicates a backwards-compatible change that adds features, and an increment of the last number indicates an internal change such as a bug-fix that should not be visible to users. The automatic installation tools use these numbers to decide which version of a package to download; if a package requests a package of version number at least 2.3.1, then the tool considers version 2.5.0 an acceptable substitute but not version 3.4.2.

All of these systems are in some sense optimistic, because they all let a tool decide to substitute one version of a package for another, and there is no way to know for certain that the program making the request doesn't depend on some hidden behavior that differs between the two. Still, in practice this system seems to work out, since most programs are not so deeply tied to the inner workings of libraries they depend on that changes to those libraries will break them.

3. Goals

In 2003 we decided to build a "CPAN for PLT Scheme" called PLaneT[1]. We wanted our design to keep or improve on the good features of existing component systems while removing as many of the undesirable properties of those approaches as we could. Specifically, we wanted to make it very easy for PLaneT to automatically retrieve and install packages and recursively satisfy dependencies with the best available packages, while giving package developers as much flexibility as possible. We also wanted to make it easy for programmers to find available packages and incorporate them into their own programs, and easy for users of those programs to install the packages they needed. More specifically:

- We wanted programmers to be able to find available libraries easily and should be able to correctly incorporate them into programs without having to know much about PLaneT. We also wanted PLaneT's more advanced features to have a gentle learning curve.
- We wanted users to be able to correctly use programs that rely on PLaneT libraries without being aware that those libraries, or PLaneT itself, existed. Ideally, we wanted whether or not a program relies on a library to be an implementation detail.

Moreover, and perhaps most importantly, we wanted to get rid of the *statefulness* inherent to other package management systems. With CPAN, for instance, every available package might or might not be installed on any particular user's computer, so a program that relies on a particular CPAN package might or might not work for that user depending on the global state defined by which packages are installed. If that state does not have the necessary packages, then the user or a setup script has to do what amounts to a **set!** that updates the state. Just as global variables make it hard to reason about whether a particular code snippet will work, we hypothesized that the statefulness of package installations make it more difficult than necessary to use component deployment systems. We wanted to eliminate that problem to the extent possible.

One non-goal we had for PLaneT was making the client able to manage the user's overall computing environment, such as man-

[1] After considering the name "CSPAN" briefly, we decided on the name PLaneT due to the fact that it implied a global network, it contained the letters P, L, and T in the correct order, and it turned out that John Clements had coincidentally already designed a logo fit the name perfectly.

aging global system configuration files or installing shared C libraries into standard locations. PLaneT is intended to help PLT Scheme programmers share PLT Scheme libraries effectively, and we can accomplish that goal much more simply if we assume that the PLaneT client can have absolute control over its domain. That isn't to say that PLaneT code cannot interact with C libraries — in fact, thanks to Barzilay's foreign interface [3] it is often quite easy to write a PLaneT package that interacts with a C library — but distributing and installing those C libraries on all the platforms PLT Scheme supports is not a problem PLaneT is intended to solve.

Our design for PLaneT consists of a client and a server: the client satisfies programs' requests for packages by asking for a suitable package file from the server, and the server determines the best package to serve for each request.

4. The PLaneT server

Most of the rest of this paper concerns the design of the PLaneT client, but there is something to be said about the server as well. PLaneT relies on a single, centralized repository located at `http://planet.plt-scheme.org/`. That site is the ultimate source for all PLaneT packages: it contains the master list of all packages that are available, and it is also responsible for answering clients' package requests. Naturally, that means it has to decide which package to return in response to these requests. Furthermore, since the server is centralized, it is responsible for deciding what packages can be published at all.

4.1 Versioning policy

As we discussed in section 2, PLaneT, or any other automatic component deployment system, needs a policy to make tractable the decision of what packages are compatible with what other packages. The policy we have chosen is essentially a stricter variant of Debian's de facto versioning policy or RubyGems' Rational Versioning Policy, with the difference that PLaneT distinguishes between the *underlying* version of a library, which the author may choose arbitrarily, and the *package* version, which PLaneT assigns. Because of this split, we can control the policy for package version numbers without demanding that package authors conform to our numbering policies for the "real" version numbers of their packages.

The package version of any package consists of two integers: the *major* version number and the *minor* version number (which we will abbreviate *major.minor* in descriptions, though technically version numbers 1.1, 1.10, and 1.100 are all distinct). The first version of a package is always 1.0, the next backwards compatible version is always 1.1, and then 1.2, and on up, incrementing the minor version number but keeping the major version number constant. If a new version breaks compatibility, it gets the next major version and gets minor version 0, so the first version that breaks backwards compatibility with the original package is always 2.0. The pattern is absolute: compatible upgrades increment the minor version only, and incompatible upgrades increment the major version and reset the minor version.

This makes it very easy for the PLaneT server to identify the best package to send to a client in response to any request. For instance, as of this writing the PLaneT repository contains four versions of the galore.plt package with package versions 1.0, 2.0, 3.0, 3.1, 3.2, and 3.3. If a client requests a package compatible with galore.plt version 3.0, the server can easily determine that 3.3 is the right package version to use for that request, based only on the version numbers. Similarly if a client requests a package compatible with version 2.0, then it knows to respond with version 2.0 even though more recent versions of the package are available, since the developer has indicated that those versions are not backwards-compatible.

The policy's effectiveness depends on the conjecture that package maintainers' notions of backwards compatibility correspond to actual backwards compatibility in users' programs. While it is easy to come up with hypothetical scenarios in which the conjecture would be false, it seems to hold nearly always in practice, and the fact that is a more conservative version of strategies already used in successful component deployment systems gives us more assurance that our policy represents a reasonable trade-off.

4.2 Package quality control

As maintainers of the PLaneT repository, we are responsible for deciding which submitted packages to include into the repository. We decided early on that our policy for quality control should be to accept all submitted packages. We decided this for a few reasons. First, we wanted to make it as easy as possible for authors to submit packages, because after all a package repository is only useful if it contains packages. All the nontrivial quality-control metrics we could think of would either entail too much work to scale effectively orimpose barriers to submission that we thought were unacceptably high. For instance, we considered only accepting packages that provided a test suite whose tests all passed, but we decided that would discourage programmers from submitting packages without first writing large test suites in some test-suite notation that we would have to devise; this seemed too discouraging. (Of course we *want* programmers to write large, high-quality test suites for their packages, and many of them do; but *mandating* those test suites seemed overly burdensome.) Second, we suspected that low-quality packages didn't need any special weeding out, since no one would want to use them or suggest that others use them; meanwhile high-quality packages would naturally float to the top without any help.

As for malicious packages, after a similar thought process we decided that there was nothing technical we could reasonably do that would stop a determined programmers from publishing a package that intentionally did harm, and that throwing up technical hurdles would likely do more harm than good by offering users a false sense of security and malicious programmers a challenge to try and beat. But again, we suspected that the community of package users could probably address this better by reputation: bad packages and their authors would be warned against and good packages and their authors would be promoted.

In short, after some consideration we decided that trying to perform any kind of serious quality control on incoming packages amounted to an attempt at a technical solution for a social problem, so we opted to let social forces solve it instead. This solution is the same solution used by the other component systems we studied, which gave us confidence that our decision was workable.

5. The client as a language feature

The PLaneT client works by hooking in to the guts of PLT Scheme's module system. In PLT Scheme, programmers can write modules that depend on other modules in several ways. For instance, the following code:

```
(module circle-lib mzscheme
  (require (file "database.ss")) ; to use run-sql-query
  (require (lib "math.ss")) ; to use π

  (define query "SELECT radius FROM circles")
  (define areas
    (map (lambda (r) (* pi r r)) (run-sql-query q)))

  (provide areas))
```

defines a module named *circle-lib* in the default *mzscheme* language. The first thing mzscheme does when loading *circle-lib* (either in response to a direct request from the user or because some other module has required it) is to process all of its requirements. In this case there are two, (**require** (**file** "database.ss")) and (**require** (**lib** "math.ss")), each of which use a different variant of the **require** form: the (**require** (**file** "database.ss")) variant loads the module in the file database.ss in the same directory as the source code of *circle-lib*, and (**require** (**lib** "math.ss")) loads the module in the file math.ss located in PLT Scheme's standard library.

PLaneT integrates into this system by simply adding another new require form. It looks like this:

```
(require (planet "htmlprag.ss" ("neil" "htmlprag.plt" 1 3)))
```

This declaration tells mzscheme that the module that contains it depends on the module found in file htmlprag.ss from the package published by PLaneT user neil called htmlprag.plt, and that only packages with major version 1 and minor version at least 3 are acceptable.

When mzscheme processes this **require** statement, it asks the PLaneT client for the path to a matching package, in which it will look for the file the user wanted (htmlprag.ss in this case). The client keeps installed packages in its own private cache, with each package version installed in its own separate subdirectory. It consults this cache in response to these requests; due to the PLaneT package version numbering strategy, it does not need to go over the network to determine if it already has an installed a package that satisfies the request. If it already has a suitable package installed, it just returns a path do that installation; if it does not, it contacts the central PLaneT server, puts in its request, and installs the package the server returns; the process of installing that package recursively fetches and installs any other packages that might also be necessary.

This strategy goes a long way towards solving the statefulness problem we explained in section 3, for the simple reason that we can think of the statefulness problem as a simpler variant of the problem modules were designed to solve. In the absence of modules (or an equivalent to modules), programmers have to write what amount to control programs that tell the underlying Scheme system how to load definitions in such a way that the top level is set to an appropriate state before running the main program — a problem that is especially bad for compilation [6], but that causes annoyance even without considering compilation at all. The module system addresses that top-level statefulness problem by making modules explicitly state their requirements rather than relying on context, and making it mzscheme's responsibility to figure out how to satisfy those requirements; from this perspective PLaneT just extends the solution to that statefulness problem to address the package-installation statefulness problem as well[2].

The connection between packages and the **require** statement also has a few pleasant side effects. First, it makes it easy for programmers to understand: PLT Scheme programmers need to learn how to use the module system anyway, so using PLaneT just means learning one new twist on a familiar concept, not having to learn how to use an entirely new feature. Second, including package dependence information directly in the source code means that there is no need for a PLaneT programmer to write a separate metadata file indicating which packages a program relies on. PLaneT is the only component deployment system we are aware of with this property.

[2] There is one important exception where statefulness shines though: if a computer that is not connected to the network runs a program that requires a PLaneT package, then that program might or might not succeed depending on whether or not the PLaneT client has already installed a suitable version of the required package. If it has, then the requirement will succeed; otherwise it will fail and signal an error. Of course there isn't very much the PLaneT client can do to prevent this stateful behavior, but it does mean that PLaneT works best for computers that are usually online.

6. Tool support

Since the PLaneT client is a part of the language of PLT Scheme, tools that work with PLT Scheme code need to be able to work with PLaneT packages just as naturally as they work with any other construct in the language. This is particularly important for DrScheme, the PLT Scheme development environment [4], because it provides programmers with several development tools that PLT Scheme programmers expect to work consistently regardless of which features their programs use. For instance, it provides a syntax checker that verifies proper syntax and draws arrows between bound variables and their binding occurrences, a syntactic coverage checker that verifies that a program has actually executed every code path, and several debuggers and error tracers. Making PLaneT a language extension rather than an add-on tool places an extra burden on us here, since it means programmers will expect all the language tools to work seamlessly with PLaneT requires. For this reason, we have tried to make tool support for PLaneT as automatic as possible.

In doing so, we are helped by the fact that PLaneT is incorporated into Scheme rather than some other language, because due to macros Scheme tools cannot make very strong assumptions about the source code they will process. Similarly, in PLT Scheme tools must be generic in processing **require** statements because a macro-like facility exists for them as well: even in fully-expanded code, a program may not assume that a **require** statement of the form (**require** *expr*) has any particular semantics, because *expr* has not itself been "expanded" into a special value representing a particular module instantiation. To get that value, the tool must pass *expr* to a special function called the module name resolver, which is the only function that is entitled to say how a particular require form maps to a target module. PLaneT is nothing but an extension to the module name resolver that downloads and installs packages in the process of computing this mapping; since tools that care about the meaning of **require** statements have to go through the module name resolver anyway, they automatically inherit PLaneT's behavior.

This has made DrScheme's tools easy to adapt to PLaneT, and in fact almost none of them required any modification. Figure 1 shows some of DrScheme's tools in action on a small PLaneT program: the first screenshot shows the syntax check tool correctly identifying bindings that originate in a PLaneT package, and the second shows PLaneT working with DrScheme's error-tracing and debugging tools simultaneously. None of these tools needed to be altered at all to work correctly with PLaneT.

We did have to make some changes to a few tools for a couple of reasons. The first was that some tools were only designed to work on particular require forms, or made assumptions about the way that libraries would be laid out that turned out to be too strong. For instance, both the Help Desk and the compilation manager assumed that all programs were installed somewhere within the paths that comprise PLT Scheme's standard library; PLaneT installs downloaded code in other locations, which caused those tools to fail until we fixed them. In principle, this category of tool change was just fixing bugs that had been in these tools all along, though only exposed by PLaneT's new use patterns.

The second reason we had to modify some tools was that the generic action that they took for all libraries didn't really make sense in the context of PLaneT packages, so we had to add special cases. For instance, we had to add code to the Setup PLT installation management tool so that it would treat multiple installed versions of the same package specially. Also, DrScheme also includes a module browser that shows a program's requirements graph; we modified that tool to allow users to hide PLaneT requirements in that display as a special case. The module browser worked with-

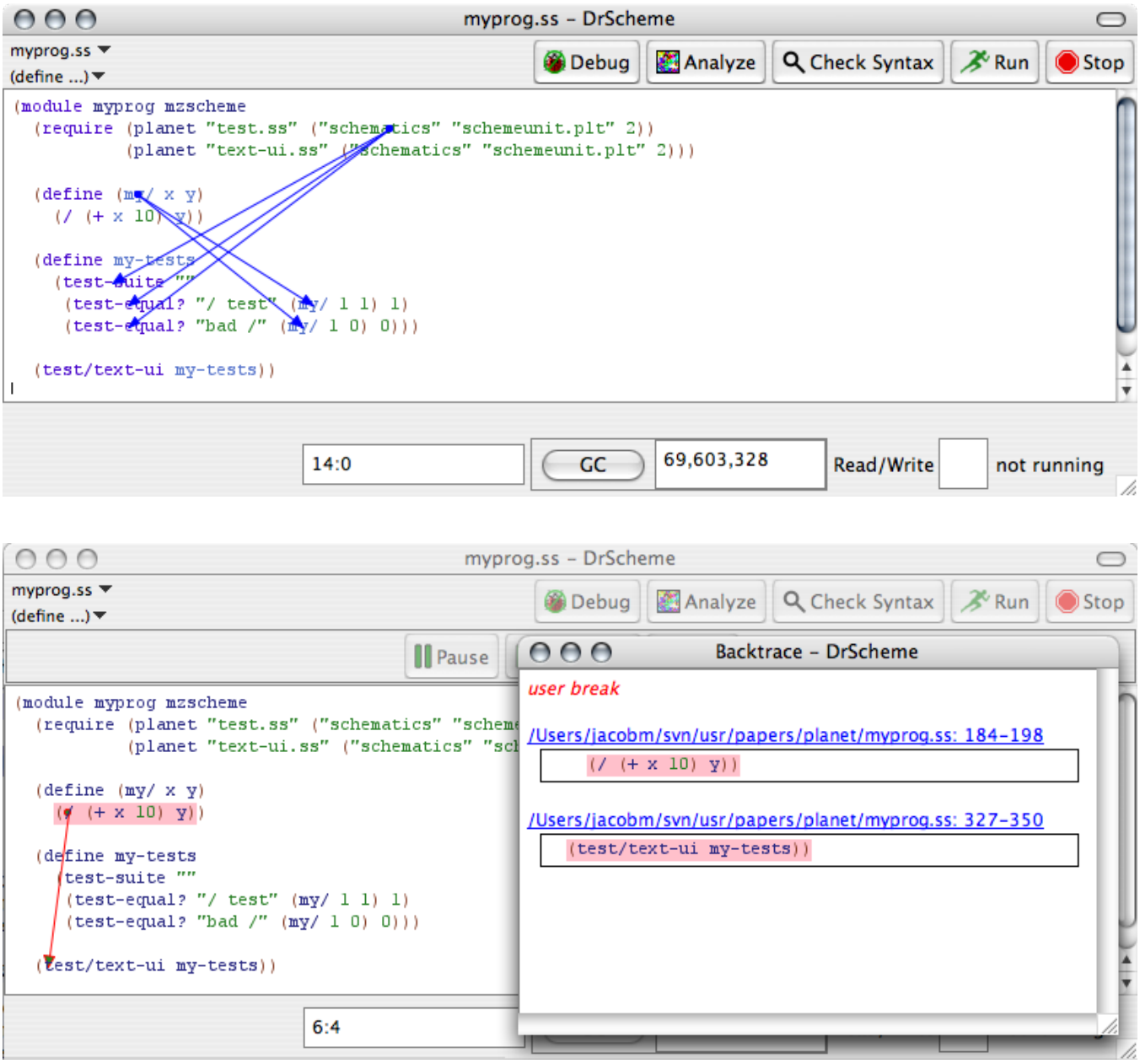


Figure 1. Two tools interacting seamlessly with PLaneT packages

out that modification, but we found that the extra requirements that PLaneT packages introduced transitively tended to add a lot of noise without being very useful. Figure 2 illustrates the impact this special case had on the quality of the module browser's output.

7. A few complications

We think PLaneT's version-naming policy works fairly well: it is simple to implement, simple to understand, and easy enough to use that if you don't understand it at all you'll probably do the right thing anyway. But of course versioning is never that quite that simple, and we have had to make a few tweaks to the system to make sure it didn't cause subtle and difficult-to-debug problems. Three of these problems, which we call the "bad update" problem, the "magic upgrade" problem and the module state problem, deserve special attention.

The bad update problem

If a program relies on buggy, undocumented or otherwise subject-to-change behavior in a package (for instance because it works around a bug), then it may break in the presence of upgrades that PLaneT thinks of as compatible (for instance the eventual bug fix). We expect these cases to be the exception, not the rule (if we thought these were the common case then we wouldn't have added versioning to PLaneT at all), but they could still represent major problems for programmers unlucky enough to have to deal with them.

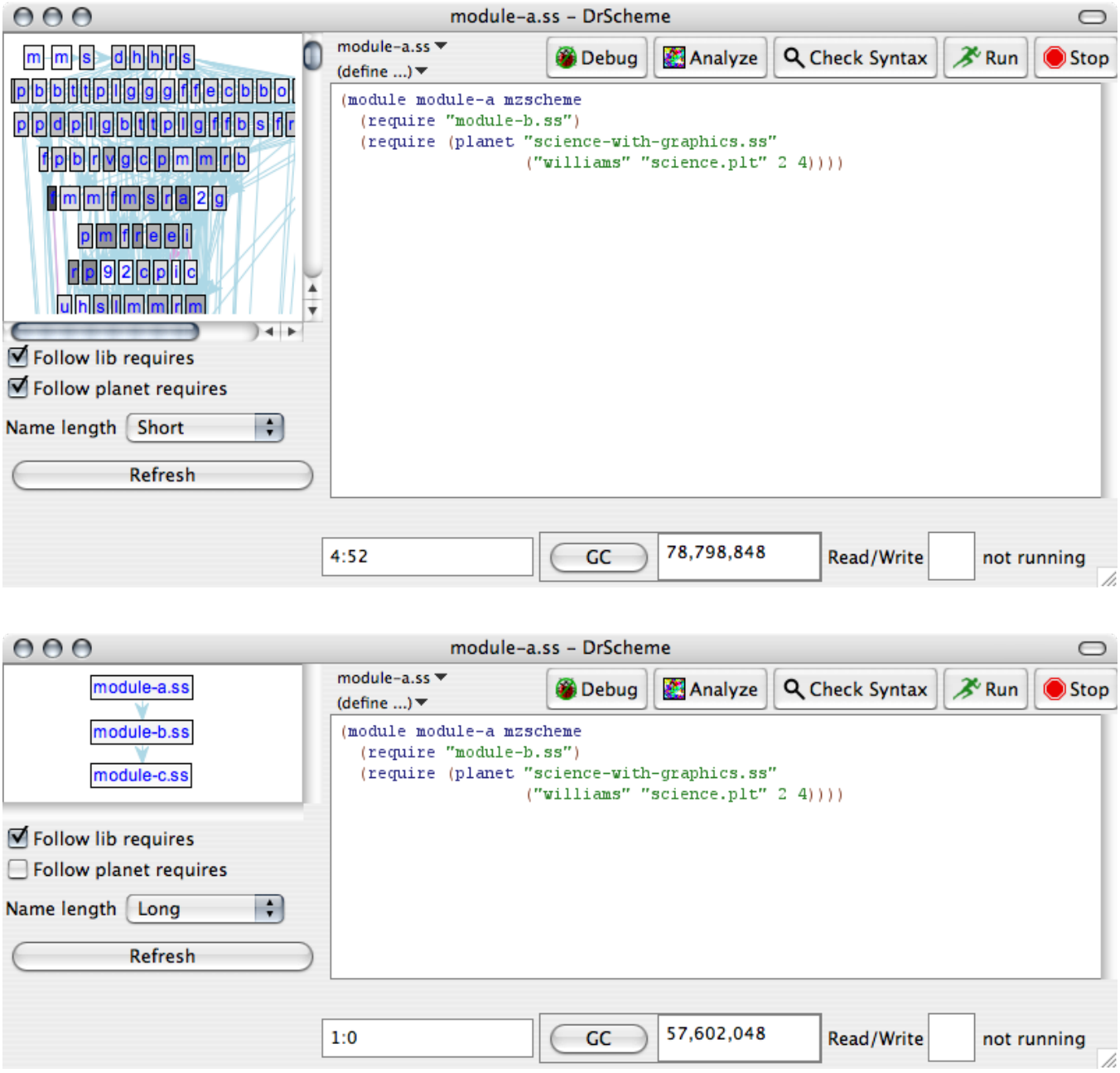


Figure 2. Sometimes special cases are important: the module browser displaying all requirements (above), and hiding PLaneT requirements (below)

To help those programmers cope, we decided early on to follow the lead set by the other component deployment systems we studied and make it possible to write a PLaneT **require** statement that asks for a more specific package than just anything backwards-compatible with a named package. In general, a PLaneT require specification may specify any range for a package's minor version (but not the major version, since two different major versions of a package don't have the same interface in general); for instance the package specifier ("soegaard" "galore.plt" 3 (0 1))) indicates that either package version 3.0 or version 3.1 is acceptable, but no others. In fact the normal way of writing a package specifier — ("soegaard" "galore.plt" 3 0) — is just syntactic sugar for ("soegaard" "galore.plt" 3 (0 +inf.0)). Similarly, the equality form ("soegaard" "galore.plt" 3 (= 0)) is just syntactic sugar for ("soegaard" "galore.plt" 3 (0 0)). We hope that programmers will not need to use these more general facilities often, but we expect that they are occasionally very useful to have around.

The magic upgrade problem

A subtler and stranger behavior that we had to be careful not to allow in PLaneT is what we call the magic upgrade problem, in which a user could run a program and end up changing the behavior of seemingly unrelated programs. If, in satisfying a requirement for a particular package specification, the PLaneT client always were to always look for the most recent locally-installed package that meets the criteria, then the following situation could arise: suppose a user has a program P that requires package A with package version 1.0 or later as part of its implementation, and has package A version 1.1 installed locally to satisfy that requirement. If package A releases a new version, 1.2, and then the user runs program Q which requires package A version 1.2 or later, then PLaneT must install package A version 1.2 locally. But now, if the user runs the original program P, *its* behavior will change because instead of using version 1.1, the client can now supply it with the newer package version 1.2. This might change program P's behavior in unpredictable ways, which is bad because according to our design principles, the user isn't supposed to have to know that package A even exists, much less that P and Q happen both to use different versions of it and so running Q might "magically upgrade" P without any warning!

Rather than allow this to happen, we have introduced a layer of indirection. When a program requires a PLaneT package for the first time, the PLaneT client remembers which particular package it told mzscheme to use to satisfy that requirement. Whenever the same program asks again, it returns the same path, regardless of whether newer packages are available. We call these associations "links" and a particular module's set of links its "linkage"; the links are collectively stored in the PLaneT client's "linkage table." The PLaneT client adds a link to a module every time it resolves a request for that module, and a module's linkage persists until the user explicitly clears it.

Module state conflicts

Another problem that can come up when two versions of a package are installed at the same time is that they may inadvertently interfere with each other because they both behave in similar ways but have different module state. In PLT Scheme, a module has only one copy of its mutable state, no matter how many times other modules require it — this allows a module to establish its own whole-program invariants or to regulate access to unique, non-duplicatable resources on the system. For instance, suppose a library author writes a module that maintains some internal state in the course of its duties:

```
(module db mzscheme

  (define *dbfile* "my-database.db")

  (define num-items 0)


  (define (write-to-db datum)
    (with-output-to-file *dbfile*
      (lambda ()
        (display datum)
        (set! num-items (add1 num-items)))))


  (provide write-to-db))
```

The author of this code might reasonably expect that no matter how a program used the *db* module to write values into my-database.db, *num-items* would remain a faithful count of the number of items there (assuming that the file started out empty, of course). That author would be correct, because no matter how many modules require the *db* module they will each get the same copy with shared internal state.

If the author puts this module in a PLaneT package, the same behavior applies. However, that behavior may not be good enough anymore, because PLaneT allows multiple versions of the same package to run side-by-side. Suppose the developer releases a new version of the *db* package:

```
;; new db module for PLaneT package version 1.1
(module db mzscheme

  (define *dbfile* "my-database.db")
  (define num-items 0)
  (define (write-to-db datum) ... [as before])
  (define (read-from-db)
    (with-input-from-file *dbfile*
      (lambda ()
        (do ((ret null)
             (i 0 (+ i 1)))
            ((= i num-items) ret)
          (set! ret (cons (read) ret))))))


  (provide write-to-db read-from-db))
```

Again, the developer might expect that *num-items* is always a true count of the number of items written into my-database.db. But it might not be the case anymore: from the solutions to the bad update problem and the magic upgrade problem, we know that for a variety of reasons different libraries within the same program might end up loading the same package with two different versions because of packages insisting on particular package versions or because of different modules in the same program getting different links. In particular, that means that a single program might use the *db* modules from package version 1.0 and package version 1.1 at the same time, and as far as mzscheme is concerned those are two separate modules with completely distinct states. If that were to happen, writes that went through *db* version 1.0 would not be reflected in version 1.1's counter, possibly leading to a corrupt database file or even a program crash if the program called *read-from-db*.

We expect that most packages will not exhibit problems like this, because most programming libraries do not rely on invariants between module-level state and system state in the way the *db* module does. However, we also expect that for the modules that do rely on those invariants, this problem could easily be a source of inscrutable bugs. Therefore our design takes the conservative approach and by default we do not allow multiple versions of the same library to be loaded at the same time, unless explicitly allowed.

Considering all of these issues together, we have arrived at the following final policy for clients resolving a PLaneT require statement for a particular program and a particular package.

1. PLaneT first decides what package to use to satisfy the request:
   (a) If the program has a link in the linkage table to a particular version of the package being requested, then PLaneT always uses that package.
   (b) If the program does not have a link, then PLaneT obtains a package that satisfies the request either from its local cache or from the central PLaneT server.
2. If PLaneT decides on a version of a package, and another version of that same package is already loaded (not just installed but actually running) in some other part of the program, then PLaneT signals an error unless the package itself explicitly tells PLaneT that simultaneous loading should be allowed. Otherwise it uses the package it has decided on to satisfy the request.

8. Evaluation

Up to this point we have described PLaneT as we originally designed it. PLaneT has been in active use for over two years, and in that time we have gained a lot of experience with how well our design decisions have worked out in practice. Generally we have found that our design works well, though we have identified a few problems and oversights, both with its technical design and with its ease-of-use features (which we consider to be at least as important).

First the good: PLaneT has quickly become many developers' default way to distribute PLT Scheme libraries, and over the course of two years we have received 115 packages and nearly 300,000 lines of Scheme code, compared to about 700,000 of Scheme code in PLT Scheme's standard library (which includes among other things the entire DrScheme editor implementation). These user-contributed libraries have been very useful for adding important functionality that PLT Scheme itself does not include. For instance, PLaneT currently holds implementations of several sophisticated standard algorthims, three different database interfaces, two unit-testing frameworks, and several bindings to popular native libraries. PLT Scheme itself does not come with much built-in support for any of these, so the easy availability of PLaneT packages has been a big help to people writing programs that need those. (Schematics' unit-testing package SchemeUnit [16], packaged as schemeunit.plt, is the single most popular package on PLaneT and is required by over a quarter of the other available packages.) Its utility is also demonstrated by its usage statistics: PLaneT packages have been download 22475 times (as of the moment of this writing), an average of about 30 a day since PLaneT was launched (and an average of about 50 a day in the last year).

We have also found that integration of package installation into the **require** statement has had the effect we had hoped it would, that programmers would have fewer qualms about using packages than they would otherwise. Our evidence is anecdotal, but we have found that code snippets on the PLT Scheme users' mailing list and in other programming fora have frequently included PLaneT require statements, without even necessarily calling attention to that fact; before PLaneT it was not common at all for people to post code that relied on nonstandard libraries.

It is harder to say how successful our package version numbering strategy has been. We have not yet heard of any problems resulting from backwards incompatibilities (with one arguable exception as we discuss below), but we have also found that nearly half of all packages (54 out of 115) have only ever released one version, and most (72 of 115) had released only one or two versions. This is not in itself alarming — on CPAN, for instance, most packages only ever release one version as well — but none of our version-compatibility machinery is relevant to packages with only one version, so the fact that it has not given us problems is not very informative.

Another area where we are still unsure of whether our policy was correct is our choice not to perform any quality control on incoming packages. While we still believe that we cannot and should not try to verify that every submitted package is "good," we have begun to think that it may be beneficial to make a few sanity checks before accepting a package. For instance, if an author submits a new version of a package and claims the new version is backwards-compatible, we cannot verify that absolutely but we can at least make sure that it provides all the same modules the old package did, and that none of those modules fails to provide a name that was provided before. This does impose an additional hurdle for package authors to clear, but in practice it seems that the only packages that fail this test have obvious packaging errors that authors would like to know about.

There have been a few design problems we have had to negotiate. One significant problem was that as new versions of PLT Scheme were released, they introduced backwards incompatibilities, and packages written for one version did not work later ones. Similarly, code written for newer versions of PLT Scheme used features that were not available in older versions, which could cause problems for users of those older versions who tried to download the new packages. This is the same problem that PLaneT's version numbering scheme tries to address, of course, but since PLT Scheme is not itself a PLaneT package we could not reuse that scheme directly. Furthermore, the PLT Scheme distribution does occasionally introduce large, generally backwards-incompatible changes in its releases, big events that define a new "series", but much more regularly it introduces more minor incompatibilities. These incompatibilities are obscure enough that we thought PLaneT packages would nearly never be affected by them, so we did not want to make package authors release new versions of their packages in response to each one. Considering these, we decided on the following policy: when an author submits a package, we associate a PLT Scheme series with it (currently there are two such series, the "2xx" series and the "3xx" series), which places that package in the 2xx or 3xx repository. By default we assume that any PLT Scheme version in a series can use any package in its corresponding repository; packages can override this assumption by indicating a minimum required PLT Scheme version if necessary.

Another problem that quickly became apparent in our initial design was that we had imposed an annoying hurdle for package developers. Developers understandably want to test their packages as they will appear to users, and in particular they want to be able to require and test packages using the (**require** (**planet** $\cdots$)) form since that form is what others will use. With our original design, there was no way to do that; the only way to have your package accessible as via PLaneT was to actually submit it, so developers had to interact with their code using either a (**require** (**file** $\cdots$)) or a (**require** (**lib** $\cdots$)) statement instead. This caused many problems and considerable frustration. Our first attempt to solve the problem was to allow programmers to install a package file directly to their PLaneT clients' local caches without going through the server. This helped but did not eliminate the problem, since programmers still had to create and install a package every time they wanted to run their tests. Based on the response to that, we arrived at our current solution: programmers can create "development links" that tell the PLaneT client to look for a given package version in an arbitrary directory of the programmer's choice. Since we added development links we have not had any more problems with programmers not being able to test their packages adequately.

9. Conclusions and related work

With two years of experience, we feel confident now in concluding that PLaneT has basically met the goals we set out for it: it is easy to use for package authors, developers who use PLaneT packages, and end users who don't want to think about PLaneT packages at all. We attribute this ease of use to our unusual inclusion of package management features into the language itself, which has allowed us to do away with the concept of an uninstalled package from the programmer's perspective. While to our knowledge that feature is unique, PLaneT otherwise borrows heavily from previously-existing package distribution systems and language-specific library archives.

As we have already discussed, we have taken insight for our design from the designs of CPAN, Debian Linux's `dpkg` system and Ruby's RubyGems system; and there are several other package management systems that also bear some level of similarity to PLaneT. Many Linux distributions come with automatic package managers (such as Gentoo's `portage` system [15] and the RPM Package Manager [7]), and there have also been a few implementations of completely automatic application-launching systems, *i.e.* application package managers that eliminate the concept of an uninstalled application: Zero Install [8] and klik [9] are examples of this kind of system. Furthermore there are many language-specific component and component-deployment systems for other language implementations; examples of these include Chicken Scheme's eggs system [17], Michael Schinz' `scsh` package manager [13], and the Common Lisp `asdf` and asdf-install system [2].

We hope with PLaneT we have achieved a synthesis of all these ideas into a single coherent system that is as natural and as easy to use as possible. We believe that every programming language should make it simple for one programmer to reuse another programmer's code; and we believe a key part of that is just making sure that programmers have a standard way to access each others' programs. As we have shown, moving package management into the language's core is a powerful way to achieve that goal.

Acknowledgements

We would like to acknowledge Robby Findler, Dave Herman, Ryan Culpepper, Noel Welsh, and the anonymous reviewers for their valuable comments about PLaneT in general and this paper. Thanks also to the National Science Foundation for supporting this work.

References

[1] Elaine Ashton and Jarkko Hietaniemi. Cpan frequently asked questions. `http://www.cpan.org/misc/cpan-faq.html`.

[2] Dan Barlow and Edi Weitz. ASDF-Install. `http://common-lisp.net/project/asdf-install/`.

[3] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

[4] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

[5] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://www.plt-scheme.org/software/mzscheme/`.

[6] Matthew Flatt. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

[7] Eric Foster-Johnson. *Red Hat RPM Guide*. Red Hat, 2003.

[8] Thomas Leonard. The zero install system, 2006. `http://zero-install.sourceforge.net/`.

[9] Simon Peter. klik, 2006. `http://klik.atekon.de/`.

[10] Matt Pietrek. Avoiding DLL hell: Introducing application metadata in the Microsoft .NET framework. *MSDN Magazine*, October 2000. Available online: `http://msdn.microsoft.com/msdnmag/issues/1000/metadata/`.

[11] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 2001.

[12] RubyGems user guide, 2006. `http://www.rubygems.org/read/book/1`.

[13] Michael Schinz. A proposal for scsh packages, August 2005. `http://lampwww.epfl.ch/~schinz/scsh_packages/install-lib.pdf`.

[14] Gustavo Noronha Silva. *APT HOWTO*, 2005. `http://www.debian.org/doc/manuals/apt-howto/apt-howto.en.pdf`.

[15] Sven Vermeulen *et al*. A portage introduction, 2006. `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1`.

[16] Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of the Third Workshop on Scheme and Functional Programming*, 2002.

[17] Felix L. Winkelmann. Eggs unlimited, 2006. `http://www.call-with-current-continuation.org/eggs/`.