Scheme 2003

Proceedings of the Fourth Workshop on Scheme and Functional Programming

Boston, Massachusetts

Matthew Flatt, editor

UUCS-03-023

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 7, 2003

Abstract

This report contains the papers presented at the Fourth Workshop on Scheme and Functional Programming. The purpose of the Scheme Workshop is to discuss experience with and future developments of the Scheme programming language—including the future of Scheme standardization—as well as general aspects of computer science loosely centered on the general theme of Scheme.

Preface

This report contains the papers presented at the Fourth Workshop on Scheme and Functional Programming, which was held on November 7, 2003 in Boston, Massachusetts.

Out of twelve papers submitted in response to the call for papers, the program committee chose nine in a virtual meeting. We thank external reviewers Frederic Boussinot, Kent Dybvig, Jonathan Riehl, and Paul Steckler for their generous help. In addition to accepted papers, the committee invited three demonstration talks, which have no entry in these proceedings. (You had to be there.)

Paper submission and review were managed using the CONTINUE server—which is implemented in Scheme, of course. We would like to thank Peter Hopkins, of Brown University, who provided and supported the server. In true Scheme form, Peter used his own server to submit a paper describing the way that server is implemented.

Matthias Felleisen graciously took charge of local arrangements for the workshop. I am further indebted to Matthias, Olin Shivers, and the rest of the steering committee for advice in organizing the workshop.

Matthew Flatt, University of Utah
Workshop chair
For the program committee

Program Committee

Robert Bruce Findler, University of Chicago
Erick Gallesio, Université de Nice
Tim Hickey, Brandeis University
Mario Latendresse, FNMOC
Jeffrey Mark Siskind, Purdue University
Mitchell Wand, Northeastern University

Steering Committee

William D. Clinger, Northeastern University
Marc Feeley, University of Montreal
Matthias Felleisen, Northeastern University
Dan Friedman, Indiana University
Christian Queinnec, University Paris 6
Manuel Serrano, INRIA
Olin Shivers, Georgia Institute of Technology
Mitchell Wand, Northeastern University

Contents

PLoT Scheme 1
Alexander Friedman and Jamie Raymond

PICBIT: A Scheme System for the PIC Microcontroller 7
Marc Feeley and Danny Dubé

Dot-Scheme: A PLT Scheme FFI for the .NET framework 16
Pedro Pinto

From Python to PLT Scheme 24
Philippe Meunier and Daniel Silva

How to Add Threads to a Sequential Language Without Getting Tangled Up 30
Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard Kelsey

Unwind-protect in portable Scheme 48
Dorai Sitaram

Enabling Complex UI In Web Applications With send/suspend/dispatch 53
Peter Walton Hopkins

Well-Shaped Macros 59
Ryan Culpepper and Matthias Felleisen

Porting Scheme Programs 69
Dorai Sitaram

PLoT Scheme

Alexander Friedman and Jamie Raymond
College of Computer Science
Northeastern University
Boston, MA 02115
USA
{cozmic,raymond}@ccs.neu.edu

ABSTRACT

We present PLTplot, a plotting package for PLT Scheme. PLTplot provides a basic interface for producing common types of plots such as line and vector field plots from Scheme functions and data, an advanced interface for producing customized plot types, and support for standard curve fitting. It incorporates renderer constructors, transformers from data to its graphical representation, as values. Plots are also values. PLTplot is built as an extension on top of the third-party PLplot C library using PLT Scheme's C foreign function interface. This paper presents the core PLTplot API, examples of its use in creating basic and customized plots and fitting curves, and a discussion of its implementation.

1 INTRODUCTION

This paper describes PLTplot a plotting extension for PLT Scheme [6] based on the PLplot [5] C library. PLTplot is provided as a set of modules that provide language constructs and data types for producing plots of functions and data. The basic interface provides constructors for rendering data and functions in common forms such as points and lines. For advanced users, PLTplot provides an interface for building custom renderer constructors on top of basic drawing primitives to obtain almost any kind of desired plot.

Our motivation for producing PLTplot was to be able to do plotting from within our favorite programming language, Scheme, instead of our usual method of only using Scheme to work with the data and then calling an external program, such as Gnuplot [10], to actually produce the plots. This mechanism was tedious, especially when we only wanted a quick and dirty plot of a Scheme function or data contained in a Scheme list or vector.

To develop the package as quickly as possible, instead of creating a plot library on top of PLT Scheme's graphical toolkit, MrEd, which would have meant a lot of engineering work, we decided to reuse the heavy lifting already done by the visualization experts. We looked at existing plotting packages and libraries written in C with appropriate free source licensing on which we could build an extension for PLT Scheme using its C foreign function interface (FFI).

Initially we considered using Gnuplot, but it builds only as a monolithic executable and not as a library. Modifying it was too daunting as it has an extremely baroque codebase. We looked elsewhere and found an LGPLed plotting library called PLplot [5] that has been used as an extension by several other programming languages. PLplot is currently being developed as a Sourceforge project; it provides many low-level primitives for creating 2D and 3D graphs of all sorts. With PLplot's primitives wrapped as Scheme functions as a foundation, we created a high-level API for plotting and included some additional utilities such as curve fitting.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Alexander Friedman, Jamie Raymond

2 PLTPLOT IN ACTION

PLTplot supports plotting data and functions in many common forms such as points, lines, or vector fields. It also supports the creation of custom renderers that can be used to visualize the data in new ways. Data can be fitted to curves and the resulting functions plotted along with the original data. We illustrate these ideas with an example from physics.

2.1 Simple Plots

Figure 1 shows a plot of data [9] similar to that collected by Henry Cavendish in 1799 during his famous experiment to weigh the earth. The X axis represents time, while the Y axis represents the angle of rotation of Cavendish's torsional pendulum. The data is defined as a list of Scheme vectors, each one containing a value for time, angle of rotation, and error.

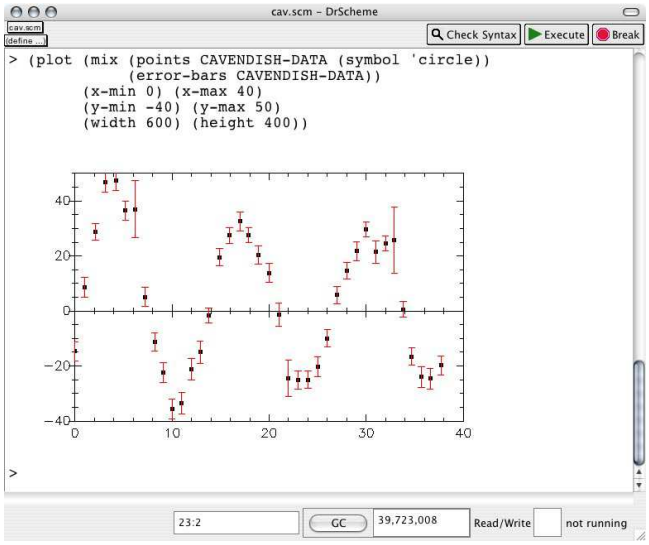
```
(define CAVENDISH-DATA
  (list (vector 0.0 -14.7 3.6)
        (vector 1.0 8.6 3.6)
        ...                      ;; omitted points
        (vector 37.7 -19.8 3.5)))
```

Before the data can be plotted, the plot module must be loaded into the environment using the following code:

```
(require (lib "plot.ss" "plot"))
```

As illustrated in Figure 1, the plot is generated from code entered at the REPL of DrScheme, PLT Scheme's programming environment. `points` and `error-bars` are constructors for values called Plot-items that represent how the data should be rendered. The call to `mix` takes the two Plot-items and composes them into a single Plot-item that `plot` can display. The code ends with a sequence of associations that instruct `plot` about the range of the X and Y axes and the dimensions of the overall plot.

Figure 1 Points and Error Bars



2.2 Curve Fitting

From the plot of angles versus time, Cavendish could sketch a curve and model it mathematically to get some parameters that he could use to compute the weight of the earth. We use PLTplot's built-in curve fitter to do this for us.

To generate a mathematically precise fitted curve, one needs to have an idea about the general form of the function represented by the data and should provide this to the curve fitter. In Cavendish's experiment, the function comes from the representation of the behavior of a torsional pendulum. The oscillation of the pendulum is modeled as an exponentially decaying sinusoidal curve given by the following equation: $f(s) = \theta_0 + ae^{\frac{-s}{\tau}} \sin(\frac{T}{2\pi s} + \phi)$. The goal is to come up with values for the constant parameters in the function so that the phenomena can be precisely modeled. In this case, the fit can give a value for T, which is used to determine the spring constant of the torsional fiber. This constant can be used to compute the universal gravitational constant – the piece Cavendish needed to compute the weight of the earth.

PLTplot fits curves to data using a public-domain implementation of the the standard Non-Linear Least Squares Fit algorithm. The user provides the fitter with the function to fit the data to, hints and names for the constant values, and the data itself. It will then produce a `fit-result` structure which contains, among other things, values for the contants and the fitted function with the computed parameters.

Figure 2 shows the code for generating the fit and producing a new plot that includes the fitted curve. To get the value of the parameter T, we select the values of the final parameters from `result` with a call to `fit-result-final-params`, shown in the code, and then inspect its output for the value.

Figure 2 Mixed Plot with Fitted Curve

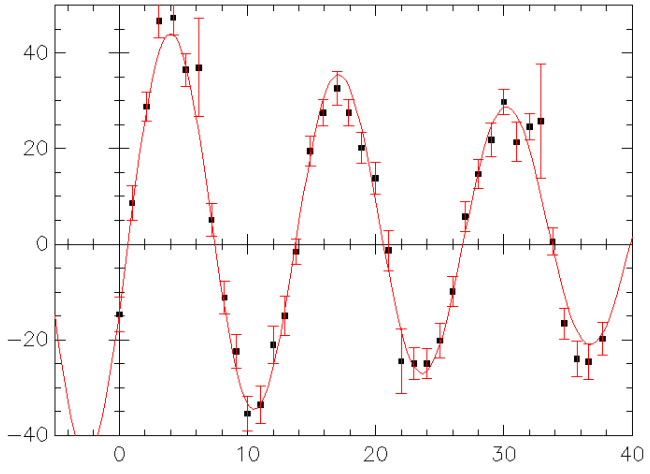
```
(require (lib "plot.ss" "plot"))

(define theta
  (lambda (s a tau phi T theta0)
    (+ theta0
       (* a
          (exp (/ s tau -1))
          (sin (+ phi (/ (* 2 pi s) T)))))))

(define result
  (fit
   theta
   ((a 40) (tau 15) (phi -.5) (T 15) (theta0 10))
   CAVENDISH-DATA))

(fit-result-final-parms result)

(plot (mix
        (points CAVENDISH-DATA)
        (error-bars CAVENDISH-DATA)
        (line (fit-result-function result)))
      (x-min -5) (x-max 40)
      (y-min -40) (y-max 50))
```



2.3 Complex Plots

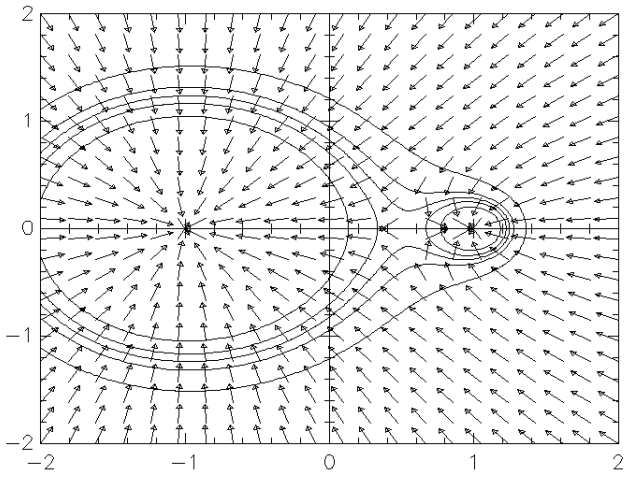
Besides operations to produce simple point and line plots, PLTplot also supports more complex plot operations. In this next example we have an equation which represents the gravitational potential of two bodies having unequal masses located near each other. We plot this equation as both a set of contours and as a vector field with several non-default options to better visualize the results. For the contour part of the plot, we manually set the contour levels and the number of times the function is sampled. For the vector part, we numerically compute the gradient of the function and reduce the number of vectors displayed to 22 and change the style of the vectors to normalized. The code and resulting plot are shown in Figure 3.

Figure 3 Contour and Vector Field

```
(require (lib "plot.ss" "plot"))

(define gravitational-potential
  (lambda (x y)
    (-
     (/ -1
        (sqrt (+ (sqr (add1 x)) (sqr y))))
     (/ 1/10
        (sqrt (+ (sqr (sub1 x)) (sqr y)))))))

(plot
 (mix (contour
       gravitational-potential
       (levels '(-0.7 -0.8 -0.85 -0.9 -1))
       (samples 100))
      (field
       (gradient
         (lambda (x y)
           (* -1 (gravitational-potential x y))))
       (samples 22) (style 'normalized)))
 (x-min -2) (x-max 2)
 (y-min -2) (y-max 2))
```



3 CORE PLTPLOT

PLTplot is meant to be easy to use and extensible. The functionality is naturally split into two levels: a basic level, which provides a set of useful constructors that allow creation of common types of plots, and an advanced level, which allows the creation of custom renderer constructors. The API for core PLTplot is shown in Figure 4.

PLTplot, in addition to being a library for PLT Scheme, is a little language for plotting. The idea is to keep the process of plotting a function or data set as simple as possible with as little decoration as necessary. This is a cognitive simplification for the casual PLTplot user, if not also a syntactic one. The special form `plot`, for instance, takes a Plot-item (constructed to display the data or function in a particular way, such as a line or only as points) followed by a possibly empty sequence of attribute-value associations. If `plot` were a function, these associations would have to be specially constructed as Scheme values, for examples as lists, which would necessitate decoration that is irrelevant to specifying the specific features of the generated plot. Other forms are provided for similar reasons.

3.1 Basic Plotting

The fundamental datatype in PLTplot is the Plot-item. A Plot-item is a transformer that acts on a view to produce a visual representation of the data and options that the Plot-item was constructed with. An interesting and useful feature of the constructed values that Plot-items produce is that they are functionally composable. This is practically used to produce multiple renderings of the same data or different data on the resulting view of the plot. Plot-items are displayed using the `plot` special form. `plot` takes a Plot-item and some optional parameters for how the data should be viewed and produces an object of the `2d-view%` class which DrScheme displays.

Plot-items are constructed according to the definitions shown in Figure 4. Consider the `line` constructor. It consumes a function of one argument and some options and produces a transformer that knows how to draw the line that the function represents. Its options are a sequence of keyword-value associations. Some possible `line-option`s include `(samples number)` and `(width number)`, specifying the number of times the function is sampled and the width of the line, respectively. Each of the other constructors have similar sets of options, although the options are not necessarily shared between them. For example, the `samples` option for a line has no meaning for constructors that handle discrete data.

The other Plot-item constructors grouped with `line` in the definition of Plot-item are used for other types of plots. `points` is a constructor for data representing points. `error-bars` is a constructor for data representing points associated with error values. `shade` is a constructor for a 3D function in which the height at a particular point would be displayed with a particular color. `contour` is likewise a constructor for a 3D function that produces contour lines at the default or user-specified levels. And `field` is a constructor for a function that represents a vector field.

The `mix` constructor generates a new Plot-item from two or more existing Plot-items. It is used to combine multiple

Figure 4 Core PLTplot API

Data Definitions

```
A Plot-item is one of
 (line 2dfunction line-option*)
 (points (list-of (vector number number))
         point-option*)
 (error-bars
    (list-of (vector number number number))
    error-bar-option*)
 (shade 3dfunction shade-option*)
 (contour 3dfunction countour-option*)
 (field R2->R2function field-option*)

 (mix Plot-item Plot-item+)
 (custom (2d-view% -> void))

*-option is: (symbol TST)
  where TST is any Scheme type

A 2dfunction is (number -> number)

A 3dfunction is (number number -> number)

A R2->R2function is one of
  ((vector number number) ->
   (vector number number))
  (gradient 3d-function)

fit-result is a structure:
  (make-fit-result ... (list-of number) ...
                   (number* -> number))

2d-view% is the class of the displayed plot.
```

Forms

```
(plot Plot-item (symbol number)*)

(fit (number* -> number)
     ((symbol number)*)
     (list-of
      (vector number [number] number number)))

(define-plot-type name data-name view-name
                       ((option value)*)
                  body)
```

Procedures

```
(fit-result-final-params fit-result) ->
  (list-of number)
(fit-result-function fit-result) ->
  (number* -> number)
... additional fit-result selectors elided ...
```

items into a single one for plotting. For example in Figure 2 `mix` was used to combine a plot of points, error bars, and the best-fit curve for the same set of data.

The final Plot-item constructor, `custom`, gives the user the ability to draw plots on the view that are not possible with the other provided constructors. Programming at this level gives the user direct control over the graphics drawn on the plot object, constructed with the `2d-view%` class.

3.2 Custom Plotting

The `2d-view%` class provides access to drawing primitives. It includes many methods for doing things such as drawing lines of particular colors, filling polygons with particular patterns, and more complex operations such as rendering data as contours.

Suppose that we wanted to plot some data as a bar chart. There is no bar chart constructor, but since we can draw directly on the plot via the `custom` constructor we can create a bar with minimal effort.

First we develop a procedure that draws a single bar on a `2d-view%` object.

```
; draw an individual bar
(define (draw-bar x-position width height view)
  (let ((x1 (- x-position (/ width 2)))
        (x2 (+ x-position (/ width 2))))
   (send view fill
         `(,x1 ,x1 ,x2 ,x2)
         `(0 ,height ,height 0))))
```

Then we develop another procedure that when applied to an object of type `2d-view%` would draw all of the data, represented as a list of two-element lists, using `draw-bar` on the view.

```
; size of each bar
(define BAR-WIDTH .75)

; draw a bar chart on the view
(define (my-bar-chart 2dview)
  (send 2dview set-line-color 'red)
  (for-each
   (lambda (bar)
     (draw-bar (car bar) BAR-WIDTH
               (cadr bar) 2dview))
   '((1 5) (2 3) (3 5) (4 9) (5 8)))) ; the data
```

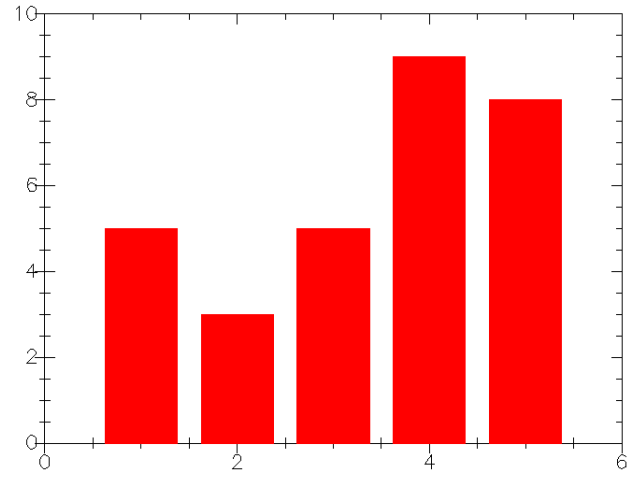
We then create the plot using the `plot` form, wrapping `my-bar-chart` with the `custom` constructor.

```
(plot (custom my-bar-chart)
      (x-min 0) (y-min 0) (x-max 6) (y-max 10))
```

The results are shown in Figure 5. The output is plain but useful. We could enhance the appearance of this chart using other provided primitives. For example, the bars could be drawn with borders, the axes given labels, etc.

While we now get the desired bar chart, we have to change the data within the `my-bar-chart` procedure each time we want a new chart. We would like to abstract over the code for `my-bar-chart` to create a generic constructor similar to the built-in ones. To manage this we use a provided special form, `define-plot-type`, which is provided in the module

Figure 5 Custom Bar Chart



`plot-extend.ss`. It takes a name for the new plot type, a name for the data, a name for the view, an optional list of fields to extract from the view, and a set of options with default values. It produces a Plot-item constructor. We can now generalize the above function as follows:

```
(require (lib "plot-extend.ss" "plot"))

(define-plot-type bar-chart
  data 2dview [(color 'red) (bar-width .75)]
    (begin
      (send 2dview set-line-color color)
      (for-each
       (lambda (bar) (draw-bar (car bar) bar-width
                               (cadr bar) 2dview))
       data)))
```

Our original data definition for Plot-item can now be augmented with the following:

```
(bar-chart (list-of (list number number))
           [(color symbol) (bar-width number)])
```

Plotting the above data with blue bars would look like:

```
(plot
  (bar-chart
    '((1 5) (2 3) (3 5) (4 9) (5 8))
    (color 'blue))
  (x-min 0) (y-min 0) (x-max 6) (y-max 10))
```

3.3 Curve Fitting API

Any scientific plotting package would be lacking if it did not include a curve fitter. PLTplot provides one through the use of the `fit` form. `fit` must be applied to a function (a representation of the model), the names and guesses for the parameters of the model, and the data itself. The guesses for the parameters are hints to the curve fitting algorithm to help it to converge. For many simple model functions the guesses can all be set to 1.

The data is a list of vectors. Each vector represents a data point. The first one or, optionally, two elements of a data vector are the values for the independent variable(s). The last two elements are the value for the dependent variable and the weight of its error. If the errors are all the same, they can be left as the default value 1.

The result of fitting a function is a `fit-result` structure. The structure contains the final values for the parameters and the fitted function. These are accessed with the selectors `fit-result-final-params` and `fit-result-function`, respectively. The structure contains other useful values as well that elided here for space reasons but are fully described in the PLTplot documentation.

4 IMPLEMENTATION

PLTplot is built on top of a PLT Scheme extension that uses the PLplot C library for its plotting and math primitives. To create the interface to PLplot, we used PLT Scheme's C FFI to build a module of Scheme procedures that map to low-level C functions. In general, the mapping was straight forward – most types map directly, and Scheme lists are easily turned into C arrays.

Displayed plots are objects created from the `2d-view%` class. This class is derived from PLT Scheme's `image-snip%` class which gives us, essentially for free, plots as first-class values. In addition `2d-view%` acts as a wrapper around the low level module. It provides some error checking and enforces some implied invariants in the C library.

It was important that PLTplot work on the major platforms that PLT Scheme supports: Windows, Unix, and Mac OS X. To achieve this we used a customized build process for the underlying PLplot library that was simplified by using `mzc` – the PLT Scheme C compiler – which generated shared libraries for each platform.

5 RELATED WORK

There is a tradition in the Scheme community of embedding languages in Scheme for drawing and graphics. Brian Beckman makes the case that Scheme is a good choice as a core language and presents an embedding for doing interactive graphics [2]. Jean-François Rotgé embedded a language for doing 3D algebraic geometry modeling [8]. We know of no other published work describing embedding a plotting language in Scheme.

One of the most widely used packages for generating plots for scientific publication is Gnuplot, which takes primitives for plotting and merges them with an ad-hoc programming language. As typically happens in these cases the language grows from something simple, say for only manipulating columns of data, to being a full fledged programming language. Gnuplot is certainly an instance of Greenpun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp." [3]. What you would rather have is the marriage of a well-documented, non-buggy implementation of an expressive programming language with full-fledged plotting capabilities.

Some popular language implementations, like Guile and Python, provide extensions for Gnuplot. This is a step forward because now one can use his or her favorite program-

ming language for manipulating data and setting some plot options before shipping it all over to Gnuplot to be plotted. However, the interface for these systems relies on values in their programming languages being translated into Gnuplot-ready input and shipped as strings to an out-of-process Gnuplot instance. For numerical data this works reasonably well, but if functions are to be plotted, they must be written directly in Gnuplot syntax as strings, requiring the user to learn another language, or be parsed and transformed into Gnuplot syntax, requiring considerable development effort.

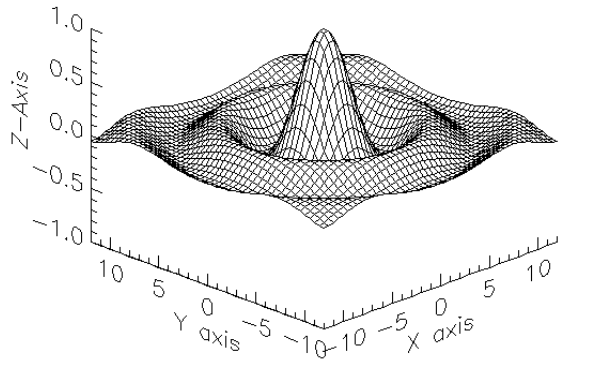
The idea behind our integration was to join the well-specified and well-documented PLT Scheme with a relatively low-level library for drawing scientific plots, PLplot. We then could handle both data and options in the implementation in a rich way rather than as plain strings. We then provided a higher-level API on top of that. Other platforms that have integrated PLplot only provide the low-level interface. These include C, C++, Fortran-77, Tcl/TK, and Java among others.

Ocaml has a plotting package called Ocamlplot [1], which was built as an extension to libplot, part of GNU plotutils [4]. libplot renders 2-D vector graphics in a variety of formats and can be used to create scientific plots but only with much effort by the developer. Ocamlplot does not provide a higher-level API like PLTplot does. For example, there is no abstraction for line plots or vector plots. Instead the user is required to build them from scratch and provide his own abstractions using the lowest-level primitives. There is also no notion of first class plots as plots are output to files in specific graphic formats.

6 CONCLUSION

This paper presented core PLTplot, which is a subset of what PLTplot provides. In addition to the core 2D API illustrated in this paper, PLTplot also provides an analogous API for generating 3D plots, an example of which is seen in Figure 6.

Figure 6 $\frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}$



PLTplot is still new and many additions are planned for the future. With the publication of this paper, the first release of PLTplot will be made available through the PLT Scheme Libraries and Extensions website [7]. Currently plots are only output as `2dview%` objects. One addition we hope to make soon is the ability to save plots in different formats including Postscript. We also plan on developing a separate plotting environment which will have its own interface for easily generating, saving, and printing plots.

7 ACKNOWLEDGMENTS

The authors would like thank Matthias Felleisen for his advice during the design of PLTplot and subsequent comments on this paper. They also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Andrieu, O. Ocamlplot [online]. Available from World Wide Web: `http://ocamlplot.sourceforge.net`.

[2] Beckman, B. A scheme for little languages in interactive graphics. `Software-Practice and Experience 21` (Feb. 1991), 187–207.

[3] Greenspun, P. Greenspun's Tenth Rule of Programming [online]. Available from World Wide Web: `http://philip.greenspun.com/research/`.

[4] Maier, R., and Tufillaro, N. Gnu plotutils [online]. Available from World Wide Web: `http://www.gnu.org/software/plotutils`.

[5] PLplot core development team. PLplot [online]. Available from World Wide Web: `http://plplot.sourceforge.net`.

[6] PLT. PLT Scheme [online]. Available from World Wide Web: `http://www.plt-scheme.org`.

[7] PLT. Plt scheme libraries and extensions [online]. Available from World Wide Web: `http://www.cs.utah.edu/plt/develop/`.

[8] Rotgé, J.-F. SGDL-Scheme: a high level algorithmic language for projective solid modeling programming. In `Proceedings of the Scheme and Functional Programming 2000 Workshop` (Montreal, Canada, Sept. 2000), pp. 31–34.

[9] Vrable, M. Cavendish data [online]. Available from World Wide Web: `http://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html`.

[10] Williams, T., and Hecking, L. Gnuplot [online]. Available from World Wide Web: `http://www.gnuplot.info`.

PICBIT: A Scheme System for the PIC Microcontroller

Marc Feeley / Université de Montréal
Danny Dubé / Université Laval

ABSTRACT

This paper explains the design of the PICBIT R^{4}RS Scheme system which specifically targets the PIC microcontroller family. The PIC is a popular inexpensive single-chip microcontroller for very compact embedded systems that has a ROM on the chip and a very small RAM. The main challenge is fitting the Scheme heap in only 2 kilobytes of RAM while still allowing useful applications to be run. PICBIT uses a novel compact (24 bit) object representation suited for such an environment and an optimizing compiler and byte-code interpreter that uses RAM frugally. Some experimental measurements are provided to assess the performance of the system.

1 INTRODUCTION

The Scheme programming language is a small yet powerful high-level programming language. This makes it appealing for applications that require sophisticated processing in a small package, for example mobile robot navigation software and remote sensors.

There are several implementations of Scheme that require a small memory footprint relative to the total memory of their target execution environment. A full-featured Scheme system with an extended library on a workstation may require from one to ten megabytes of memory to run a simple program (for instance MzScheme v205 on Linux has a 2.3 megabyte footprint). At the other extreme, the BIT system [1] which was designed for microcontroller applications requires 22 kilobytes of memory on the 68HC11 microcontroller for a simple program with the complete R^{4}RS library (minus file I/O). This paper describes a new system, PICBIT, which is inspired from our BIT system and specifically designed for the PIC microcontroller family which has even tighter memory constraints.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Marc Feeley and Danny Dubé.

2 THE PIC MICROCONTROLLER

The PIC is one of the most popular single-chip microcontroller families for low-power very-compact embedded systems [6]. There is a wide range of models available offering RISC-like instruction sets of 3 different complexities (12, 14, or 16 bit wide instructions), chip sizes, number of I/O pins, execution speed, on-chip memory and price. Table 1 lists the characteristics of a few models from the smallest to the largest currently available.

BIT was originally designed for embedded platforms with 10 to 30 kilobytes of total memory. We did not distinguish read-only (ROM) and read-write (RAM) memory, so it was equally important to have a compact object representation, a compact program encoding and a compact runtime. Moreover the design of the byte-code interpreter and libraries favors compactness of code over execution speed, which is a problem for some control applications requiring more computational power. The limited range of integers (-16384 to 16383) is also awkward. Finally, the incremental garbage collector used in BIT causes a further slowdown in order to meet real-time execution constraints [2].

Due to the extremely small RAM of the PIC, it is necessary to distinguish what needs to go in RAM and what can go in ROM. Table 1 shows that for the PIC there is an order of magnitude more ROM than RAM. This means that the compactness of the object representation must be the primary objective. The compactness of the program encoding and runtime is much less of an issue, and can be traded-off for a more compact object representation and speedier byte-code interpreter. Finally, we think it is probably acceptable to use a nonincremental garbage collector, even for soft real-time applications, because the heap is so small.

We call our Scheme system PICBIT to stress that the characteristics of the PIC were taken into account in its design. However the system is implemented in C and it should be easy to port to other microcontrollers with similar memory constraints. We chose to target the "larger" PIC models with 2 kilobytes of RAM or more (such as the PIC18F6520) because we believed that this was the smallest RAM for doing useful work. Our aim was to create a practical system

Model	Pins	MIPS	ROM	RAM	Price
PIC12C508	8	1	512 × 12 bits	25 × 8 bits	$0.90
PIC16F628	18	5	2048 × 14 bits	224 × 8 bits	$2.00
PIC18F6520	64	10	16384 × 16 bits	2048 × 8 bits	$6.50
PIC18F6720	64	6.25	65536 × 16 bits	3840 × 8 bits	$10.82

Table 1: Sample PIC microcontroller models.

that strikes a reasonable compromise between the conflicting goals of fast execution, compact programs and compact object representation.

3 OBJECT REPRESENTATION

3.1 Word Encoding

In many implementations of dynamically-typed languages all object references are encoded using words of W bits, where W is often the size of the machine's words or addresses [3]. With this approach at most 2^W references can be encoded and consequently at most 2^W objects can live at any time. Each object has its unique encoding. Since many types of objects contain object references, W also affects the size of objects and consequently the number of objects that can fit in the available memory. In principle, if the memory size and mix of live objects are known in advance, there is an optimal value for W that maximizes the number of objects that can coexist.

The 2^W object encodings can be partitioned, either statically (e.g. tag bits, encoding ranges, type tables) or dynamically (e.g. BIBOP [4]) or a combination, to map them to a particular type and representation. A representation is `direct` if the W bit word contains all the information associated with the object, e.g. a fixnum or Boolean (the meaning of "all the information" is left vague). In an `indirect` representation the W bit word contains the address in memory (or an index in a table) where auxiliary information associated with the object is stored, e.g. the fields of a pair or string. The direct representation can't be used for mutable objects because mutation must only change the state of the object, not its identity. When an indirect representation is used for immutable objects the auxiliary information can be stored in ROM because it is never modified, e.g. strings and numbers appearing as literals in the program.

Like many microcontrollers, the PIC does not use the same instructions for dereferencing a pointer to a RAM location and to a ROM location. This means that when the byte-code interpreter accesses an object it must distinguish with run time tests objects allocated in RAM and in ROM. Consequently there is no real speed penalty caused by using a different representation for RAM and ROM, and there are possibly some gains in space and time for immutable objects.

Because the PIC's ROM is relatively large and we expect the total number of immutable objects to be limited, using the indirect representation for immutable objects requires relatively little ROM space. Doing so has the advantage that we can avoid using some bits in references as tags. It means that we do not have to reserve in advance many of the 2^W object encodings for objects, such as fixnums and characters, that may never be needed by the program. The handling of integers is also simplified because there is no small vs. large distinction between integers. It is possible however that programs which manipulate many integers and/or characters will use more RAM space if these objects are not preallocated in ROM. Any integer and character resulting from a computation that was not preallocated in ROM will have to be allocated in RAM and multiple copies might coexist. Interning these objects is not an interesting approach because the required tables would consume precious RAM space or an expensive sweep of the heap would be needed. To lessen the problem, a small range of integers can be preallocated in ROM (for example all the encodings that are "unused" after the compiler has assigned encodings to all the program literals and the maximum number of RAM objects).

3.2 Choice of Word and Object Size

For PICBIT we decided that to get simple and time-efficient byte-code interpreter and garbage collector all objects in RAM had to be the same size and that this size had to be a multiple of 8 bits (the PIC cannot easily access bit fields). Variable size objects would either cause fragmentation of the RAM, which is to be avoided due to its small size, or require a compacting garbage collector, which are either space- or time-inefficient when compared to the mark-sweep algorithm that can be used with same size objects. We considered using 24 bits and 32 bits per object in RAM, which means no more than 682 and 512 objects respectively can fit in a 2 kilobyte RAM (the actual number is less because the RAM must also store the global variables, C stack, and possibly other internal tables needed by the runtime). Since some encodings are needed for objects in ROM, W must be at least 10, to fully use the RAM, and no more than 12 or 16, to fit two object references in an object (to represent pairs).

With $W = 10$, a 32 bit object could contain three object references. This is an appealing proposition for compactly representing linked data structures such as binary search tree nodes, special association lists and continuations that the interpreter might use profitably. Unfortunately many bits would go unused for pairs, which are a fairly common data type. Moreover, $W = 10$ leaves only a few hundred encodings for objects in ROM. This would preclude running programs that

1. contain too many constant data-structures (the system would run out of encodings);

2. maintain tables of integers (integers would fill the RAM).

But these are the kind of programs that seem likely for microcontroller applications (think for example of byte buffers, state transition tables, and navigation data). We decided

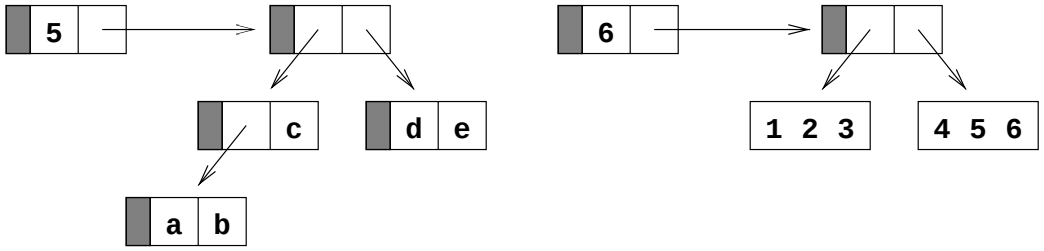


Figure 1: Object representation of the vector `#(a b c d e)` and the string `"123456"`. To improve readability some of the details have been omitted, for example the "`a`" is really the object encoding of the symbol `a`. The gray area corresponds to the two tag bits. Note that string leaves use all of the 24 bits to store 3 characters.

that 24 bit objects and $W = 11$ was a more forgiving compromise, leaving at least 1366 ($2^W - 682$) encodings for ROM objects.

It would be interesting to perform an experiment for every combination of design choice. However, creating a working implementation for one combination requires considerable effort. Moreover, it is far from obvious how we could automate the creation of working implementations for various spaces of design choice. There are complex interactions between the representation of objects, the code that implements the operations on the objects, the GC, and the parts of the compiler that are dependent on these design choices.

3.3 Representation Details

For simplicity and because we think ROM usage is not an important concern for the PIC, we did not choose to represent RAM and ROM objects differently.

All objects are represented indirectly. That is, they all are allocated in the heap (RAM or ROM) and they are accessed through pointers. Objects are divided in three fields: a two bit tag, which is used to encode type information, and two 11 bit fields. No type information is put on the references to objects. The purpose of each of the two 11 bit fields (X and Y) depends on the type:

`00` ⇒ `Pair.` X and Y are object references for the `car` and `cdr`.

`01` ⇒ `Symbol.` X and Y are object references for the name of the symbol (a string) and the next symbol in the symbol table. Note that the name is not necessary in a program that does not convert between strings and symbols. PICBIT does not currently perform this optimization.

`10` ⇒ `Procedure.` X is used to distinguish the three types of procedures based on the constant C (number of lambdas in the program) which is determined by the compiler, and the constant P (number of Scheme primitive procedures provided by the runtime, such as `cons` and `null?`, but not `append` and `map` which are defined in the library, a Scheme source file):

- $0 \leq X < C \Rightarrow$ `Closure.`X is the entry point of the procedure (raw integer) and Y is an object reference to the environment (the set of nonglobal free variables, represented with an improper list).
- $C \leq X < C + P \Rightarrow$ `Primitive.`X is the entry point of the procedure (raw integer) and Y is irrelevant.
- $X = C + P \Rightarrow$ `Reified continuation.`X is an object reference to a continuation object and Y is irrelevant. A continuation object is a special improper list of the form `(`r p `.` e`)`, where r is the return address (raw integer), p is the parent continuation object and e is an improper list environment containing the continuation's live free variables.

The runtime and P are never modified even when some primitive procedures are not needed by the compiled program.

`11` ⇒ One of `vector`, `string`, `integer`, `character`, `Boolean` or `empty list`. X is a raw integer that determines the specific type. For integer, character, Boolean and empty list, X is less than 36 and Y is also a raw integer. For the integer type, 5 bits from X and 11 from Y combine to form a 16 bit signed integer value. For the vector type, $36 \leq X < 1024$ and $X - 36$ is the vector's length. For the string type, $1024 \leq X < 2048$. To allow a logarithmic time access to the elements of vectors and strings, Y is an object reference to a balanced tree of the elements. A special case for small vectors (length 0 and 1) and small strings (length 0, 1, and 2) stores the elements directly in Y (and possibly 5 bits of X for strings of length 2). Figure 1 gives an example of how vectors and strings are represented. Note that the leaves of strings pack 3 characters.

4 GARBAGE COLLECTION

The mark-sweep collector we implemented uses the Deutsch-Schorr-Waite marking algorithm [7]. This algorithm can traverse a linked data structure without using an auxiliary stack, by reversing the links as it traverses the data structure (we call such reversed links "back pointers"). Conceptually two bits of state are attached to each node. The `mark` bit indicates that the node has been visited. The `stage` bit indicates which of the two links has been reversed.[1] When the

[1] In fact, a `mark` bit should be attached to each node instead of two bits since the stage bit is meaningless when the mark bit is not set.

marking algorithm returns to a node as part of its backtracking process using the current back pointer (i.e. the "top of the stack"), it uses the stage bit to know which of the two fields contains the next back pointer. The content of this field must be restored to its original value, and if it is the first field then the second field must be processed in turn.

These bits of information cannot be stored explicitly in the nodes because all 24 bits are used. The mark bit is instead stored in a bit vector elsewhere in the RAM (this means the maximal number of objects in a 2 kilobyte RAM is really 655, leaving 1393 encodings for ROM objects).

We use the following trick for implementing the stage bit. The address in the back pointer has been shifted left by one position and the least significant bit is used to indicate which field in the "parent" object is currently reversed. This approach works for the following reason. Note that stage bits are only needed for nodes that are part of the chain of reversed links. Since there are more ROM encodings than RAM encodings and a back pointer can only point to RAM, we can use a bit of the back pointer to store the stage bit. A back pointer contains the stage bit of the node that it points to.

One complication is the traversal of the nodes that don't follow the uniform layout (with two tag bits), such as the leaves of strings that contain raw integers. Note that references to these nodes only occur in a specific type of "enclosing" object. This is an invariant that is preserved by the runtime system. It is thus possible to track this information during the marking phase because the only way to reach an object is by going through that specific type of enclosing object. For example, the GC knows that it has reached the leaf of a string because the node that refers to it is an internal string tree node just above the leaves (this information is contained in the type bits of that node).

After the marking phase, the whole heap is scanned to link the unmarked nodes into the free list. Allocation removes one node at a time from the free list, and the GC process is repeated when the free list is exhausted.

5 BYTE-CODE INTERPRETER

The BIT system's byte-code interpreter is relatively slow compared to other Scheme interpreters on the same platform. One important contributor to this poor performance is the management of intermediate results. The evaluation "stack" where intermediate results are saved is actually implemented with a list and every evaluation, including that of constants and variables, requires the allocation of a pair to link it to the stack. This puts a lot of pressure on the garbage collector, which is not particularly efficient because it is incremental. Moreover, continuations are not safe-for-space.

To avoid these problems and introduce more opportunities for optimization by the compiler, we designed a register-based virtual machine for PICBIT. Registers can be used to store intermediate results and to pass arguments to procedures. It is only when these registers are insufficient that values must be saved on an evaluation stack. We still use a linked representation for the stack, because reserving a contiguous section of RAM for this purpose would either be wasteful (stack section too large) or risk stack overflows (stack section too small). Note that we don't have the option of growing the stack and heap toward each other, because our garbage collector does not compact the heap. Substantial changes to the object representation would be needed to permit compaction.

The virtual machine has six registers containing object references: `Acc`, `Arg1`, `Arg2`, `Arg3`, `Env`, and `Cont`. `Acc` is a general purpose accumulator, and it contains the result when returning to a continuation. `Arg1`, `Arg2`, and `Arg3` are general purpose and also used for passing arguments to procedures. If there are more than three arguments, `Arg3` contains a list of the third argument and above. `Env` contains the current environment (represented as an improper list). `Cont` contains a reference to a continuation object (which as explained above contains a return address, a reference to the parent continuation object and an environment containing the continuation's live free variables). There are also the registers `PC` (program counter) and `NbArgs` (number of arguments) that hold raw integers. When calling an inlined primitive procedure (such as `cons` and `null?`, but not `apply`), all registers except `Acc` and `PC` are unchanged by the call. For other calls, all registers are caller-save except for `Cont` which is callee-save.

Most virtual machine instructions have register operands (source and/or destination). Below is a brief list of the instructions to give an idea of the virtual machine's size and capabilities. We do not explain all the instruction variants in detail.

`CST` *addr*, r ⇒ Load a constant into register r.

`MOV`[`S`] r_1, r_2 ⇒ Store r_1 into r_2.

`REF`(`G`|[`T`][`B`]) i, r ⇒ Read the global or lexical variable at position i and store it into r.

`SET`(`G`|[`T`][`B`]) r, i ⇒ Store r into the global or lexical variable at position i.

`PUSH` r_1, r_2 ⇒ Construct the pair `(cons` r_1 r_2`)` and store it into r_2.

`POP` r_1[, r_2] ⇒ Store `(car` r_1`)` into r_2 and store `(cdr` r_1`)` into r_1.

`RECV`[`T`] n ⇒ Construct the environment of a procedure with n parameters and store it into `Env`. This is normally the first instruction of a procedure.

`MEM`[`T`][`B`] r ⇒ Construct the pair `(cons` r_1 `Env)` and store it into `Env`.

`DROP` n ⇒ Remove the n first pairs of the environment in `Env`.

`CLOS` n ⇒ Construct a closure from n (entry point) and `Acc` and store it into `Acc`.

`CALL` n ⇒ Set `NbArgs` to n and invoke the procedure in `Acc`. Register `Cont` is not modified (the instruction does not construct a new continuation).

`PRIM` i ⇒ Inline call to primitive procedure i.

`RET` ⇒ Return to the continuation in `Cont`.

`JUMPF` r, $addr$ $\Rightarrow$ If r is false, branch to address $addr$.

`JUMP` $addr$ $\Rightarrow$ Branch to address $addr$.

`SAVE` n $\Rightarrow$ Construct a continuation from n (return point), `Cont`, and `Env` and store it into `Cont`.

`END` $\Rightarrow$ Terminate the execution of the virtual machine.

6 COMPILER

PICBIT's general compilation approach is similar to the one used in the BIT compiler. A whole-program analysis of the program combined with the Scheme library is performed and then the compiler generates a pair of C files ("`.c`" and "`.h`"). These files must be compiled along with PICBIT's runtime system (written in C) in a single C compilation so that some of the data-representation constants defined in the "`.h`" file can specialize the runtime for this program (i.e. the encoding range for RAM objects, constant closures, etc). The "`.h`" file also defines initialized tables containing the program's byte-code, constants, etc.

PICBIT's analyses, transformations and code generation are different from BIT's. In particular:

- The compiler eliminates useless variables. Both lexical and global variables are subject to elimination. Normally, useless variables are rare in programs. However, the compiler performs some transformations that turn many variables into useless ones. Namely, constant propagation and copy propagation, which replace references to variables that happen to be bound to constants and to the value of immutable variables, respectively. Variables that are not read and that are not set unsafely (e.g. mutating a yet undefined global variable) are deemed useless.

- Programs typically contain literal constant values. The compiler also handles closures with no nonglobal free variables as constants (this is possible because there is a single instance of the global environment). Note that all library procedures and typically most or all top-level user procedures can be treated like constants. This way globally defined procedures can be propagated by the compiler's transformations, often eliminating the need for the global variable they are bound to.

- The compiler eliminates dead code. This is important, because the R^{4}RS runtime library is appended to the program and the compiler must try to discard all the library procedures that are unnecessary. This also eliminates constants that are unnecessary, which avoids wasting object encodings. The dead code elimination is based on a rather simplistic test: the value of a variable that is read for a reason other than being copied into a global variable is considered to be required. In practice, the test has proved to be precise enough.

- The compiler determines which variables are live at return points, so that only those variables are saved in the continuations created. Similarly, the environments stored into closures only include variables that are needed by the body of the closures. This makes continuations and closures safe-for-space. It is particularly important for an embedded system to be safe-for-space. For example, an innocent-looking program such as the one in Figure 2 retains a considerable amount of data if the closures it generates include unnecessary variables. PICBIT has no problem executing it.

```
(define (make-list n x)
  (if (<= n 0)
      '()
      (cons x (make-list (- n 1) x))))

(define (f lst)
  (let* ((len (length lst))
         (g (lambda () len)))
    (make-list 100 g)))

(define (many-f n lst)
  (if (<= n 0)
      lst
      (many-f (- n 1) (f lst))))

(many-f 20000 (make-list 100 #f))
```

Figure 2: Program that requires the safe-for-space property.

7 EXPERIMENTAL RESULTS

To evaluate performance we use a set of six Scheme programs that were used in our previous work on BIT.

`empty` Empty program.

`thread` Small multi-threaded program that manages 3 concurrent threads with `call/cc`.

`photovore` Mobile robot control program that guides the robot towards a source of light.

`all` Program which references each Scheme library procedure once. The implementation of the Scheme library is 737 lines of Scheme code.

`earley` Earley's parser, parsing using an ambiguous grammar.

`interp` An interpreter for a Scheme subset running code to sort a list of six strings.

The `photovore` program is a realistic robotics program with soft real-time requirements that was developed for the LEGO MINDSTORMS version of BIT. The source code is given in Figure 3. The other programs are useful to determine the minimal space requirements (`empty`), the space requirements for the complete Scheme library (`all`), the space requirements for a large program (`earley` and `interp`), and to check if multi-threading implemented with `call/cc` is feasible (`thread`).

We consider `earley` and `interp` to be complex applications that are atypical for microcontrollers. Frequently, microcontroller applications are simple and control-oriented, such as `photovore`. Many implement finite state machines, which are table-driven and require little RAM. Applications that may require more RAM are those based on

```
; This program was originally developed for controlling a LEGO
; MINDSTORMS robot so that it will find a source of light on the floor
; (flashlight, candle, white paper, etc).

(define narrow-sweep 20) ; width of a narrow "sweep"
(define full-sweep 70) ; width of a full "sweep"
(define light-sensor 1) ; light sensor is at position 2
(define motor1 0) ; motor 1 is at position A
(define motor2 2) ; motor 2 is at position C

(define (start-sweep sweeps limit heading turn)
  (if (> turn 0) ; start to turn right or left
      (begin (motor-stop motor1) (motor-fwd motor2))
      (begin (motor-stop motor2) (motor-fwd motor1)))
  (sweep sweeps limit heading turn (get-reading) heading))

(define (sweep sweeps limit heading turn best-r best-h)
  (write-to-lcd heading) ; show where we are going
  (if (= heading 0) (beep)) ; mark the nominal heading
  (if (= heading limit)

      (let ((new-turn (- turn))
            (new-heading (- heading best-h) ))
        (if (< sweeps 20)
            (start-sweep (+ sweeps 1)
                         (* new-turn narrow-sweep)
                         new-heading
                         new-turn)
            ; the following call is replaced by #f in the modified version
            (start-sweep 0
                         (* new-turn full-sweep)
                         new-heading
                         new-turn)))

      (let ((reading (get-reading)))
        (if (> reading best-r) ; high value means lots of light
            (sweep sweeps limit (+ heading turn) turn reading heading)
            (sweep sweeps limit (+ heading turn) turn best-r best-h)))))

(define (get-reading)
  (- (read-active-sensor light-sensor))) ; read light sensor

(start-sweep 0 full-sweep 0 1)
```

Figure 3: The source code of the `photovore` program.

multi-threading and those involved in data processing such as acquisition, retransmission, and, particularly, encoding (e.g. compressing data before transmission).

7.1 Platforms

Two platforms were used for experiments. We used a Linux workstation with a a 733 MHz Pentium III processor and `gcc` version 2.95.4 for compiling the C program generated by PICBIT. This allowed quick turnaround for determining the minimal RAM required by each program and direct comparison with BIT.

We also built a test system out of a PIC18F6720 microcontroller clocked with a 10 MHz crystal. We chose the PIC18F6720 rather than the PIC18F6520 because the larger RAM and ROM allowed experimentation with RAM sizes above 2 kilobytes and with programs requiring more than 32 kilobytes of ROM. Note that because of its smaller size the PIC18F6520 can run 4 times faster than this (i.e. at 10 MIPS with a 40 MHz clock). In the table of results we have extrapolated the time measurements to the PIC18F6520 with a 40 MHz clock (i.e. the actual time measured on our test system is 4 times larger). The ROM of these microcontrollers is of the FLASH type that can be reprogrammed several times, making experimentation easy.

C compilation for the PIC was done using the Hi-Tech PICC-18 C compiler version 8.30 [5]. This is one of the best C compilers for the PIC18 family in terms of code generation quality. Examination of the assembler code generated revealed however some important weaknesses in the context of PICBIT. Multiplying by 3, for computing the byte address of a 24 bit cell, is done by a generic out-of-line 16 bit by 16 bit multiplication routine instead of a simple sequence of additions. Moreover, big `switch` statements (such as the byte-code dispatch) are implemented with a long code sequence which requires over 100 clock cycles. Finally, the C compiler reserves 234 bytes of RAM for internal use (e.g. intermediate results, parameters, local variables) when com-

		PICBIT			BIT	
Program	LOC	Min RAM	Byte-code	ROM req.	Min RAM	Byte-code
`empty`	0	238	963	21819	2196	1296
`photovore`	38	294	2150	23050	3272	1552
`thread`	44	415	5443	23538	2840	1744
`all`	173	240	11248	32372	2404	5479
`earley`	653	2253	19293	35329	7244	6253
`interp`	800	1123	17502	35525	4254	7794

Table 2: Space usage in bytes for each system and program.

piling the test programs. Note that we have taken care not to use recursive functions in PICBIT's runtime, so the C compiler may avoid using a general stack. We believe that a hand-coding of the system in assembler would considerably improve performance (time and RAM/ROM space) but this would be a major undertaking due to the complexity of the virtual machine and portability would clearly suffer.

7.2 Memory Usage

Each of the programs was compiled with BIT and with PICBIT on the Linux workstation. To evaluate the compactness of the code generated, we measured the size of the byte-code (this includes the table of constants and the ROM space they occupy). We also determined what was the smallest heap that could be used to execute the program without causing a heap overflow. Although program execution speed can be increased by using a larger heap it is interesting to determine what is the absolute minimum amount of RAM required. The minimum RAM is the sum of the space taken by the heap, by the GC mark bits, by the Scheme global variables, and the space that the PICC-18 C compiler reserves for internal use (i.e. 234 bytes). The space usage is given in Table 2. For each system, one column indicates the smallest amount of RAM needed and another gives the size of the byte-code. For PICBIT, the ROM space required on the PIC when compiled with the PICC-18 C compiler is also indicated.

The RAM requirements of PICBIT are quite small. It is possible to run the smaller programs with less than 512 bytes of RAM, notably `photovore` which is a realistic application. RAM requirements for PICBIT are generally much smaller than for BIT. On `earley`, which has the largest RAM requirement on both systems, PICBIT requires less than 1/3 of the RAM required by BIT. BIT requires more RAM than is available on the PIC18F6520 even for the `empty` program.

The size of the byte-code and constants is up to 3 times larger for PICBIT than for BIT. The largest programs (`earley` and `interp`) take a little more than 32 KB of ROM, so a microcontroller with more memory than the PIC18F6520 is needed. The other programs, including `all` which includes the complete Scheme library, fit in the 32 KB of ROM available on the PIC18F6520.

Under the tight constraints on RAM that we consider here, even saving space by eliminating Scheme global variables is crucial. Indeed, large programs or programs that require the inclusion of a fair part of the standard library use many global variables. Fortunately, the optimizations performed by our byte-compiler are able to remove almost

Program	In sources	After UFE	After UGE
`empty`	195	0	0
`photovore`	210	43	0
`thread`	205	92	3
`all`	195	195	1
`earley`	231	142	0
`interp`	302	238	2

Table 3: Global variables left after each program transformation.

RAM size	Total run time	Avg. GC interval	Avg. GC pause time
512	84	0.010	0.002
1024	76	0.029	0.005
1536	74	0.047	0.007
2048	74	0.066	0.009
2560	74	0.085	0.011
3072	74	0.104	0.013

Table 4: Time in seconds for various operations as a function of RAM size on the `photovore` program.

all of them. Table 3 indicates the contribution of each program transformation at eliminating global variables. The first column indicates the total number of global variables found in the user program and the library. The second one indicates how many remain after useless function elimination (UFE). The third one indicates how many remain after useless global variables have been eliminated (UGE). Clearly, considerable space would be wasted if they were kept in the executable.

7.3 Speed of Execution

Due to the virtual machine's use of dynamic memory allocation, the size of the RAM affects the overall speed of execution even for programs that don't perform explicit allocation operations. This is an important issue on a RAM constrained microcontroller such as the PIC. Garbage collections will be frequent. Moreover, PICBIT's blocking collector processes the whole heap at each collection and thereby introduces pauses in the program's execution that deteriorate the program's ability to respond to events in real-time.

We used `photovore`, a program with soft real-time requirements, to measure the speed of execution. The program was modified so that it terminates after 20 sweep iterations. A total of 2791008 byte-codes are executed. The program was run on the PIC18F6720 and an oscilloscope was used to measure the total run time, the average time between collections and the average collection pause. The measures, extrapolated to a 40 MHz PIC18F6520, are reported in Table 4.

This program has few live objects throughout its execution and all collections are evenly spaced and approximately the same duration. The total run time decreases with RAM size but the collection pauses increase in duration (because the sweep phase is proportional to the heap size). The duration of collection pauses is compatible with the soft real-time constraints of `photovore` even when the largest possible

RAM size is used. Moreover the collector consumes a reasonably small portion (12% to 20%) of the total run time, so the program has ample time to do useful work. With the larger RAM sizes the system executes over 37000 byte-codes per second.

The `earley` program was also tested to estimate the duration of collection pauses when the heap is large and nearly full of live objects. This program needs at least 2253 bytes of RAM to run. We ran the program with slightly more RAM (2560 bytes) and found that the longest collection pause is 0.063 second and the average time between collections is 0.085 second. This is acceptable for such an extreme situation. We believe this to be a strong argument that there is little need for an incremental collector in such a RAM constrained system.

To compare the execution speed with other systems we used PICBIT, BIT, and the Gambit interpreter version 3.0 on the Linux workstation to run the modified `photovore` program. PICBIT and BIT were compiled with "`-O3`" and a 3072 byte RAM was used for PICBIT, and a 128 kilobyte heap was used for BIT (note that BIT needs more than 3072 bytes to run `photovore` and PICBIT can't use more RAM than that). The Gambit interpreter used the default 512 kilobyte heap. The run time for PICBIT is 0.33 second. BIT and Gambit are respectively 3 times and 5 times faster than PICBIT. Because of its more advanced virtual machine, we expected PICBIT to be faster than BIT. After some investigation we determined that the cause was that BIT is performing an inlining of primitives that PICBIT is not doing (i.e. replacing calls to the generic "`+`" procedure in the two argument case with the byte-code for the binary addition primitive). This transformation was implemented in an `ad hoc` way in BIT (it relied on a special structure of the Scheme library). We envision a more robust transformation for PICBIT based on a whole-program analysis. Unfortunately it is not yet implemented. To estimate the performance gain that such an optimization would yield, and evaluate the raw speed of the virtual machines, `photovore`'s source code was modified to directly call the primitives. The run time for PICBIT dropped to 0.058 second, making it slightly faster than Gambit's interpreter (at 0.064 second) and roughly twice the speed of BIT (at 0.111 second). The speed of PICBIT's virtual machine is quite good, especially when the small heap is taken into account.

8 CONCLUSION

We have described PICBIT, a system intended to run Scheme programs on microcontrollers of the PIC family. Despite the PIC's severely constrained RAM, nontrivial Scheme programs can still be run on the larger PIC models. The RAM space usage and execution speed is surely not as good as can be obtained by programming the PIC in assembly language or C, but it is compact enough and fast enough to be a plausible alternative for some programs, especially when quick experimentation with various algorithms is needed. We think it is an interesting environment for compact soft real-time applications with low computational requirements, such as hobby robotics, and for teaching programming.

The main weaknesses of PICBIT are its low speed and high ROM usage. The use of a byte-code interpreter, the very compact style of the library, and the intricate object representation are all contributors to the low speed. This is a result of the design choices that strongly favor compactness. The use of a byte-code interpreter allows the microcontroller to run large programs that could not be handled if they were compiled to native code. The library makes extensive use of higher-order functions and code factorization in order to have a small footprint. Specialized first-order functions would be faster at the expense of compactness. The relatively high ROM space requirements are a bit of a disappointment. We believe that the runtime could be translated into more compact native code. Barring changes to the virtual machine, improvements to the C compiler or translation by hand to assembler appear to be the only ways to overcome this problem.

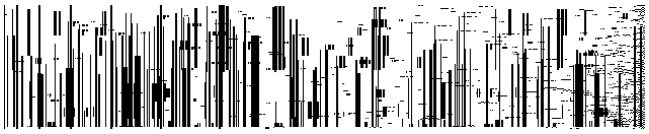


Figure 4: Heap occupancy during execution of `interp`.

PICBIT's RAM usage is the most satisfactory aspect of this work but many improvements can still be made, especially to the byte-compiler. The analyses and optimizations that it performs are relatively basic. Control-flow, type, and escape analyses could provide the necessary information for more ambitious optimizations, such as inlining of primitives, unboxing, more aggressive elimination of variables, conversion of heap allocations into static or stack allocations, stripping of useless services in the runtime, etc. The list is endless.

As an instance of future (and simple) improvement, we consider implementing a compact representation for strings and vectors intended to flatten the trees used in their representation. The representation is analogous to CDR-coding: when many consecutive cells are available, a sequence of leaves can be allocated one after the other, avoiding the need for linkage using interior nodes. The position of the objects of a sequence is obtained by pointer arithmetics relatively to a head object that is intended to indicate the presence of CDR-coding. Avoiding interior nodes both increases access speed and saves space. Figure 4 illustrates the occupancy of the heap during the execution of `interp`. The observations are taken after each garbage collection. In the graph, time grows from top to bottom. Addresses grow from left to right. A black pixel indicates the presence of a live object. There are 633 addresses in the RAM heap. The garbage collector has been triggered 122 times. One can see that the distribution of objects in the heap is very regular and does not seem to deteriorate. Clearly, there are many long sequences of free cells. This suggests that an alternative strategy for the allocation of long objects has good chances of being successful.

ACKNOWLEDGEMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and Université

Laval.

REFERENCES

[1] Danny Dubé. BIT: A very compact Scheme system for embedded applications. In Proceedings of the Workshop on Scheme and Functional Programming (Scheme 2000), pages 35–43, September 2000.

[2] Danny Dubé, Marc Feeley, and Manuel Serrano. Un GC temps réel semi-compactant. In Actes des Journés Francophones des Langages Applicatifs 1996, January 1996.

[3] David Gudeman. Representing type information in dynamically typed language. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, October 1993.

[4] Jr. Guy Steele. Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, September 1977.

[5] Hi-Tech. PICC-18 C compiler (http://www.htsoft.com/products/pic18/pic18.html).

[6] Microchip. PICmicro microcontrollers (http://www.microchip.com/1010/pline/picmicro/index.htm).

[7] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. Communications of the ACM, 10(8):501–506, August 1967.

Dot-Scheme
A PLT Scheme FFI for the .NET framework

Pedro Pinto
Blue Capital Group
105 Concord Dr., Chapel Hill, NC
27514
+ 1 919 9606042 ex17
pedro@bluecg.com

ABSTRACT

This paper presents the design and implementation of dot-scheme, a PLT Scheme Foreign Function Interface to the Microsoft .NET Platform.

Keywords

Scheme, FFI, .NET, CLR.

1. INTRODUCTION

Scarcity of library code is an often cited obstacle to the wider adoption of Scheme. Despite the number of existing Scheme implementations, or perhaps because of it, the amount of reusable code directly available to Scheme programmers is a small fraction of what is available in other languages. For this reason many Scheme implementations provide Foreign Function Interfaces (FFIs) allowing Scheme programs to use library binaries originally developed in other languages.

On Windows platforms the C Dynamic Link Library (DLL) format has traditionally been one of the most alluring targets for FFI integration, mainly because Windows's OS services are exported that way. However making a Windows C DLL available from Scheme is not easy. Windows C DLLs are not self-describing. Meta-data, such as function names, argument lists and calling conventions is not available directly from the DLL binary. This complicates the automatic generation of wrapper definitions because it forces the FFI implementer to either a write C parser to extract definitions from companion C Header files, or, alternatively, to rely on the Scheme programmer to provide the missing information.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.

Fourth Workshop on Scheme and Functional Programming.
November 7, 2003, Boston, Massachusetts, USA.
Copyright 2003 Pedro Pinto.

These issues, along with other problems including the mismatch between C's manual memory management and Scheme's garbage collection, make the use and implementation of C FFIs difficult tasks.

Recently the advent of the .NET platform [10] has provided a more attractive target for Windows FFI integration. The .NET platform includes a runtime, the Common Language Runtime or CLR, consisting of a large set of APIs covering most of the OS functionality, a virtual machine language (IL) and a just-in-time compiler capable of translating IL into native code. The CLR offers Garbage Collection services and an API for accessing the rich meta-data packaged in CLR binaries. The availability of this meta-data coupled with the CLR's reflection capabilities vastly simplifies the implementation and use of Scheme FFIs.

The remainder of this paper will illustrate this fact by examining the design and implementation of dot-scheme, a PLT Scheme [7] FFI to the CLR. Although dot-scheme currently targets only PLT Scheme, its design should be portable to any Scheme implementation that can be extended using C.

2. Presenting dot-scheme

The dot-scheme library allows the use of arbitrary CLR libraries, also called assemblies, from Scheme. Consider the following C# class:

```
using System;
public class Parrot
{
   public void SayHello(string name)
   {
      Console.WriteLine ("Hello {1}.", name);
   }
}
```

Assuming this code is compiled into an assembly, `parrot-assembly.dll`, then the `Parrot` class is available from the following Scheme code:

```
(require (lib "dot-scheme.ss" "dot-scheme"))

(import-assembly "parrot-assembly")
(:say-hello (new ::parrot) "John")

> Hello John.
```

The meaning of the Scheme code should be easy to infer. After importing the dot-scheme library bindings the program uses the `import-assembly` macro to load the `parrot-assembly` binary:

```
(import-assembly "parrot-assembly")
```

When loading is completed, `import-assembly` iterates through all the types contained in `parrot-assembly` generating appropriate wrapper bindings. The identifier `::parrot` is bound to a Scheme proxy of an instance of the CLR Type class. This instance contains meta-data associated with the `Parrot` type and can be used as an argument to the `new` function:

```
(new ::parrot)
```

The `new` function constructs an instance of the Parrot class and returns a proxy. This proxy is then used as the first argument (corresponding to the "this" pointer in C#) of the `:say-hello` function:

```
(:say-hello (new ::parrot) "John")
```

The second argument is a Scheme string. Internally `:say-hello` will extract the actual Parrot reference from the first argument, convert the Scheme string to a CLR string and then invoke the `SayHello` method on the `Parrot` object. The result of this call is a CLR string which is automatically converted to a Scheme string and returned to the top level.

In general, using a CLR assembly is fairly straightforward. The user needs only to specify which assembly to import and dot-scheme will load it and generate the appropriate wrapper code. This is true not only in this toy example but also when importing definitions from large, complex libraries such as the CLR system assemblies. For example using any one of the three-hundred plus types that comprise the CLR GUI framework is just as simple:

```
(require (lib "dot-scheme.ss" "dot-scheme"))
(import-assembly "system.windows.forms")
(::message-box:show "Hello!")
```

In general using CLR types through dot-scheme is no harder than using regular PLT Scheme modules [5]. In fact it is possible to muddle the distinction between PLT modules and CLR assemblies:

```
(module forms mzscheme
(require (lib "dot-scheme.ss"
              "dot-scheme"))
(import-assembly "system.windows.forms")
(provide (all-defined)))
```

This code above defines a PLT Scheme module named `forms`. When the module is loaded or compiled the expansion of the `import-assembly` macro creates a set of bindings within the module's scope. These bindings are exported by the declaration `(provide (all-defined))`. Assuming the module is saved in a file `forms.ss` and placed in the PLT collections path then access to the CLR GUI from Scheme simply entails:

```
(require (lib "forms.ss"))
(::message-box:show "Hello again!")
```

There is not much more to be said about using dot-scheme. Scheme's macro facilities coupled with the richness of the meta-data contained in CLR assemblies make it possible to generate Scheme wrappers from the binaries themselves. The power of this combination is apparent in the simplicity with which CLR types can be used. Perhaps even more striking though is how straightforward it is to achieve this level of integration. This should become apparent in the next section.

2.1 High-level architecture

The dot-scheme architecture can be thought of as defining two layers:

- A core layer, responsible for managing storage of CLR objects as well as CLR method dispatch. This layer is implemented in 1200 lines of Microsoft Managed C++ (MC++).
- A code generation layer, responsible for generating wrapper bindings for CLR types. These wrappers are implemented in terms of the primitives supplied by the core layer. The code generation layer is implemented in 700 lines of Scheme.

2.2 The Core Layer

Dot-scheme memory management and method dispatch are implemented in a PLT extension. A PLT extension is a DLL written in C/C++ and implementing Scheme callable functions [6]. These functions can use and create Scheme objects represented in C/C++ by the type `Scheme_Object`. At runtime Scheme programs can dynamically load extensions and use extension functions as if those functions had been defined in Scheme.

Using Microsoft's Managed C++ (MC++), a dialect of C++ which can target the CLR, it is possible to create a PLT extension that targets the CLR. Such an extension is able to use any of the CLR types as well as any of the library calls provided by the PLT runtime. From the point of view of the Scheme process that loads the extension the usage of the CLR is invisible. The extension functions initially consist of a small stub that transfers control to the CLR runtime. When the function is invoked for the first time the CLR retrieves the associated IL, translates it to machine code and replaces the method stub with a jump to the generated code.

The core layer is implemented in MC++ and so can bridge the PLT Scheme and CLR runtimes.

2.2.1 Object Representation

The first challenge faced when attempting to use .NET from Scheme is how to represent CLR data in Scheme. There are two categories of CLR data to consider. Primitive types such as integers, doubles, Booleans and strings have more or less direct equivalents in the Scheme type system and so can simply be copied to and from their Scheme counterparts. Non primitive CLR types present a more interesting problem. The CLR type system consists of a single-rooted class hierarchy where every type is a subclass of Object (even primitive types such as integers and floats can be boxed, that is their value, along with a small type descriptor, can be copied to the heap and Object references used to access it). Object life-time in the CLR is controlled by a Garbage Collector. To understand the interaction between the Scheme Garbage Collector and the CLR it is first necessary to examine the CLR's Garbage Collection strategy.

The CLR Garbage Collector is activated when the CLR heap reaches a certain size threshold. At this point the Garbage Collector thread will suspend all other threads and proceed to identify all objects that are reachable from a set of so called roots. Roots are Object references that are present in the stack, in static members or other global variables, and in CPU registers. Objects that are not reachable from this set are considered collectable and the space they occupy in the heap is considered empty. The Garbage Collector reclaims this space by moving objects to the empty space[1] and updating all changed references.

Clearly for this algorithm to work the Garbage Collector must be aware of all active Object references within a process. As a consequence it is not possible to pass Object references to code that is not running under the control of the CLR. This includes the Scheme runtime and so, to represent CLR Objects, another level of indirection is needed. Dot-scheme implements this indirection by storing references to CLR Objects in a CLR hash table using an integer key.

[1] This is a very simplified explanation. For details see [8]

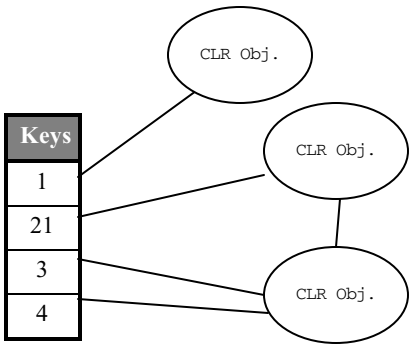


Figure 2. CLR Object management

This key can be packaged in a `Scheme_Object`, which can then be associated with a finalization callback and registered with the Scheme garbage collector. As long as the CLR Object remains in the hash table it will not be collected by the CLR (since the hash table itself is referenced by a static variable, and therefore is reachable from a root). When the Scheme runtime determines that the reference is no longer needed the finalization callback is invoked. At this point the associated integer key is used to locate and remove the CLR Object reference from the hash table. Other references to the same Object may exist either in the hash table or in other CLR objects and so the Object may continue to exist until all references go out of scope.

Notice that it will take at least two independent garbage collections for an Object to be collected, one by the Scheme runtime and one by the CLR, but otherwise this process is undistinguishable from regular Scheme garbage collection.

In terms of CLR Object representation the above is almost all that is necessary. One subtlety remains though. Consider the following classes:

```
class A
{
   virtual string SayName()
   {return "A";}

   string SayNameAgain ()
   {return "A";}
}
class B : public A
{
   override string SayName()
   {return "B";}

   string SayNameAgain()
   {return "B";}
}
```

Now assume an instance of `B` and an instance of `A` are created and references of type `A` associated to each of them:

```
A a1 = new B();
A a2 = new A();
```

What happens when `SayName` is invoked? It depends:

```
a1.SayName2() -> returns "B"
a2.SayName2() -> return "A"
```

This result should not come as a surprise. Basic dynamic dispatch is happening here. At run-time the CLR dispatches on the type of the Object pointed by `a1` and `a2`. Using the Object representation strategy described above it would be easy to wrap the `SayName` method in Scheme and still support dynamic dispatch. Assuming `a-ref` is a reference to a subtype of `A` then:

```
(say-name2 a-ref)
```

could be implemented in a PLT extension as follows[2]:

```
Scheme_Object SayName2(Scheme_Object a)
{
   A a = (A) DecodeCLRObject (a);
   return EncodeCLRObject(a.SayName2());
}
```

where `EncodeCLRObject` returns a CLR Object given an integer key (packaged in a `Scheme_Object`) and `DecodeCLRObject` stores a CLR Object and returns its key packaged as a CLR Object. In this particular case this strategy would produce the correct result. However there is another scenario. Consider the following code:

```
B bRef = new B();
A aRef = bRef;
```

what should happen when `SayNameAgain` is invoked? Again, it depends:

```
bRef.SayNameAgain () -> returns "B"
aRef.SayNameAgain () -> return "A"
```

Despite the fact that an identically named method was invoked on the same Object on both calls the actual method that is executed depends on the type of the reference used to access the Object. In contrast with Scheme where only values have types, in the CLR both an Object and the references used to manipulate it have a type. Furthermore both have a role in method dispatch, the first at runtime and the second at compile time[3]. In Scheme variables are typeless. Assume `SayNameAgain` is invoked from Scheme on a reference to an instance of `B`:

```
(say-name-again b-ref)
```

In this case problem a problem arises. What should `say-name-again` do? Should it invoke `A.SayNameAgain` or `B.SayNameAgain`? The runtime type of `b-ref` is insufficient to make this decision, so somehow it is necessary to convey the missing type information.This can be done in two ways: the reference type can be implied in the function call, for example by creating two functions, `say-name-again-A` and `say-name-again-B`, or it can be encoded in the reference itself. The latter approach is more natural for users familiar with the CLR and leads to simpler code and so is the one preferred by dot-scheme.

In terms of the structures described above only a small change is required. Instead of a CLR Object the hash table mentioned must store an object reference structure consisting of an Object and Type pair:

```
class ObjRef
{
   ObjRef (Object o, Type t)
   {obj = o; type = t;}

   Object obj;
   Type type;
}
```

The `obj` reference plays the same role as before pointing to the CLR Object of interest. The `type` reference is used to record the dispatch type associated with `obj`.

With typed references comes the need to occasionally circumvent the type system. In dot-scheme the `Cast` function changes the

[2] Throughout this paper we will present examples in C# despite the fact that the dot-scheme core layer is implemented in MC++. MC++'s syntactic noise would likely cause distraction without offering any additional insights.

[3] Note that this issue can be seen as a special case of the general problem of resolving overloaded method calls. The CLR resolves such calls at compile time based on the static types used in the method call.

dispatch type associated with a CLR reference. Implementing `Cast` is straightforward:

```
Scheme_Object Cast (Scheme_Object o,
                    Scheme_Object typeName)
{
   Type tt =
      Type.GetType(toCLRString(typeName));

   ObjRef or = DecodeCLRObject(o);

   If (tt.IsAssignableFrom(or.obj.GetType()))
      return
         EncodeCLRObject(new ObjRef(obj,tt));
   else
      throw Exception ("Invalid cast");
}
```

The code above starts by using the CLR Type class to retrieve the Type instance associated with the class named by `typeName`. After checking the validity of the cast the code then creates and returns a new `ObjRef` containing the same CLR Object but a different dispatch type.

This addition completes the description of dot-scheme's memory management strategy. To summarize, the problem of representing CLR objects in Scheme can be reduced to the implementation of the following interface:

```
Scheme_Object EncodeObjRef (ObjRef);
ObjRef DecodeObjRef (Scheme_Object ref);
void RemoveObjRef (Scheme_Object ref);
Scheme_Object Cast(Scheme_Objectm ref,
                   Scheme_Object typeName);
```

The semantics of these operations should be clear:

- `EncodeObjRef` will add the `ObjRef` passed as an argument to an internal hash table and package the associated integer key in a `Scheme_Object`. This object is then registered with the Scheme garbage collector by associating it with the `RemoveObjRef` finalization callback.
- `DecodeObjRef` will extract an integer key from the passed `Scheme_Object` and return the `ObjRef` that is associated with it.
- `RemoveObjRef` will obtain an integer key from its argument in the same way as `DecodeObjRef` but instead of returning the associated `ObjRef` it will remove it from the internal hash table.
- `Cast` creates a new `ObjRef` based on the one passed as an argument. The new `ObjRef` will be associated with the type named by `typeName`.

2.2.2 Method dispatch

Dynamic method dispatch is a complex process [2]. Consider the steps required to determine what method should be invoked by the C# code below:

```
obj.f(arg2, arg3, ... argn)
```

First, at compile time, the type of the reference `obj` is located. Within the type's scope the compiler will search for a method named `f`. Since the CLR supports method overloading several candidate methods may exist. The compiler must use the static types of the `arg2…argn` expressions to select between candidate methods. This disambiguation process is not entirely straightforward. Because of sub-typing it is still possible for several identically named methods to have signatures that are compatible with the types of the expressions `arg2…argn`. In this case the C# compiler will try to select the "most specialized" method, i.e. the method whose formal argument types are closer in the inheritance tree to the actual argument types. If a decision is possible there is still one more step. If the method found is not virtual then the compiler will emit a direct method call. If the method is virtual then final resolution is deferred until run time and the compiler simply records the index of the virtual method found. At runtime this index will be used to retrieve the corresponding method from the method table associated with the Object referenced.

In order to stay close to normal CLR semantics dot-scheme emulates this lookup process. Since Scheme is dynamically typed no reference types are available at compile time and so all the steps above have to be performed at run-time.

Dot-scheme makes use of the CLR reflection API to implement most of this process. The reflection API offers methods allowing the retrieval of type information and the invocation of methods on types which can be unknown at compile time. Every CLR Object implements a `GetType` method which returns an instance of the CLR Type class holding type information for the specific CLR Object. Using this Type Object it is possible to locate and invoke methods on the original instance. Dot-scheme relies on these capabilities to implement its dispatch algorithm. A simplified version of this algorithm is presented below. For brevity, error processing and handling of void return types are omitted.

```
ObjRef Dispatch (String methodName,
                 ObjRef self,
                 ObjRef [] args)
{
   Type [] argTypes = new Type [args.Length];
   Object []argValues = new Object[args.Length];

   For (int n = 0; n < args.Length; i++) {
      argTypes[n] = args[n].type;
      argValues[n] = args[n].obj;
   }

   MethodInfo mi =
      self.type.GetMethod (methodName,
                           argTypes);

   Object result =
      mi.Invoke(self.obj,methodName, args);

   return new ObjRef (result,
                      mi.GetReturnType());

}
```

The first step taken by `Dispatch` is extracting the types of the arguments used in the call. Note that the reference types, not the actual argument types are used. Using the extracted types `Dispatch` queries the Type Object associated with the reference on which the method call is invoked. `Type.GetMethod` is a CLR method which implements the method lookup algorithm described earlier. The result of `GetMethod` is a `MethodInfo` instance which contains meta-data associated with the method found. `MethodInfo.Invoke` is then used to execute the method. The result of this call, along with the associated reference type, is packaged in an `ObjRef` and returned.

Note that despite the relative complexity associated with method dispatching, the above code is straightforward. All the heavy lifting is done by the CLR. The reflection API is used to locate the appropriate method and, once a method is found, to construct the appropriate stack frame and transfer control to the target method.

Dot-scheme actually implements two additional variations on the dispatch code above, one for dispatching constructor calls and another for dispatching static member calls, but in essence its dispatch mechanism is captured in the code above.

2.2.3 The Core API

As mentioned earlier the core layer is implemented through a PLT Scheme extension. This extension implements the object management and dispatch mechanisms described above. In order to make its services available to the Scheme runtime the core layer exports the following Scheme constructs:

- `obj-ref`. `obj-ref` is a new Scheme type. Internally this type simply packages an integer key that can be used to retrieve ObjRef instances.
- `(cast obj-ref type-name) -> obj-ref`. The `cast` function takes as argument an `obj-ref` and a string. `obj-ref` indicates the Object reference that is the source of the cast and the string names the target type.
- `(dispatch self method-name arg…) -> result | void`. The `dispatch` function will invoke the instance method named by the `method-name` string on the CLR Object associated with the first `arg` passing the remaining parameters as arguments. Internally the implementation will examine both the `self` and `arg` parameters to determine if they correspond to `ObjRef's` or Scheme primitive types. In the first case the associated `ObjRef` instance is retrieved. In the second case a new `ObjRef` is created and the Scheme value is copied to the equivalent CLR Type. The resulting list of arguments is then passed to the MC++ dispatch call described earlier. The result of the method call, if any, is either copied to a Scheme value and immediately returned or encoded as an integer key and returned.
- `(dispatch-static type-name method-name arg …) -> result | void`. `dispatch-static` is similar to `dispatch-instance` but in this case there is no self reference. Instead the type named by the string `type-name` is searched for a static method named `method-name`.
- `(dispatch-constructor type-name arg…) -> result | void`. `dispatch-constructor` is similar to `dispatch-static` except for the fact that a constructor method is implied.

This API is sufficient to provide access to almost all of the CLRs functionality. The rest of dot-scheme's implementation simply provides syntactic sugar over these five definitions. A natural syntactic mapping is an important factor in the determining the popularity of a FFI and so the next section will examine dot-scheme's efforts in this area.

2.3 The Code Generation Layer

The Core API is all that is necessary to manipulate the Parrot class introduced earlier. The original example could be rewritten in terms of the Core API primitives:

```
(require (lib "dot-scheme-core.ss"
              "dot-scheme"))

(dispatch-instance
   (dispatch-constructor
       "ParrotAssembly, Parrot"
       "John"))
    "SayHello"
    "John")

> Hello John.
```

However this syntax is irritatingly distant from normal Scheme usage. This can easily be remedied with the help of some utilities:

```
(define-syntax import-method
    (syntax-rules ()
      ((_ ?scheme-name ?clr-name)
       (define (?scheme-name self . args)
         (apply dispatch-instance
                (cons self
                      (cons ?clr-name
                            args)))))))

(define-syntax import-type
  (syntax-rules ()
    ((_ ?scheme-name ?clr-name)
     (define ?scheme-name
       (dispatch-static "System.Type"
                        "GetType"
                        ?clr-name)))))

(define (new type-object . args)
  (apply
    (dispatch-constructor
      (dispatch-instance
        type-object
        "get_AssemblyQualifiedName")
       args)))
```

Now it is possible to write:

```
(import-type ::parrot "Parrot, ParrotAssembly")
(import-method :say-hello "SayHello")

(:say-hello (new ::parrot) "John")

> Hello John.
```

The new syntax looks like Scheme but requires typing import-statements. When importing a large number of types this may become tedious. Fortunately the CLR allows the contents of an assembly to be inspected at run-time. Using the primitives in the core layer it is possible to take advantage of the Reflection API to obtain a complete list of types in an assembly. It is then a simple matter to iterate through each type, generating the appropriate import-type/method expressions.

In fact this is almost exactly how the `import-assembly` syntax-case [3] macro works. There is a complication though. Because the CLR and Scheme use different rules for identifier naming and scoping it is not possible to map CLR names directly to Scheme.

Dot-scheme addresses these issues by renaming CLR methods and types in a way that is compatible with Scheme naming rules and hopefully produces bindings that can be easily predicted from the original CLR names. The problems addressed by this process include:

- Case sensitivity. The CLR is case sensitive while standard Scheme is not. Dot-scheme addresses this issue by mangling CLR identifiers, introducing a '-' before each upper-case character but the first (if the first character is lower-case a `-' is inserted at the beginning). Because '-' is an illegal character for CLR identifiers this mapping is isomorphic.
- Identifier scoping. In the CLR, method and type names have different scopes. It is perfectly legal to have a method `A` defined in a class `B` and a method `B` defined in a class `A`. In Scheme there is only one namespace so if both methods and types were mapped in the same way collisions could occur. Dot-scheme address this issue by prefixing type names with '::' and method names with ':'.
- Namespaces. Languages that target the CLR provide some mechanisms to segregate type definitions into namespaces. A namespace is an optional prefix for a type name. The problem faced by dot-scheme is what to do when identifiers coming from different namespaces collide. Dot-scheme's solution for this issue is very simple. For each CLR type dot-scheme will create two bindings. One of the bindings will include the namespace prefix associated with the type and the other will not. In most of the cases the Scheme programmer will use the shorter version. In case of a collision the long name can be used.
- Differences in method name scoping. In the CLR each method name is scoped to the class in which it is declared. Since dot-scheme dispatches instance method calls by searching for a method declared in the type of the first argument, instance method name collisions pose

no difficulties. However static methods present a different challenge. In this case there is no "this" pointer to reduce the scope of the method name search. Dot-scheme addresses this issue by prefixing each static method name with the type name the method is associated with.

The code generation layer in dot-scheme consists essentially of a set of macros that generate bindings according to the rules described above.

3. Future work

As presented dot-scheme allows the creation and use of CLR objects. This is, of course, only half the problem. Callbacks from the CLR to Scheme would also be very useful, in particular for implementing Graphical User Interfaces. The CLR allows the generation of code at run-time so it should be possible for dot-scheme to generate new CLR classes based on Scheme specifications. Future work will investigate this possibility. Performance issues are also likely to be visited. Currently dot-scheme's implementation favors simplicity over performance when a choice is necessary. As the system matures we expect this bias to change.

4. Conclusion

This paper supports two different goals. Ostensibly the goal has been to present the design and implementation of a Scheme FFI. A more covert but perhaps more important goal was to alert Scheme implementers to the ease with which bindings to the CLR can be added to existing Scheme implementations.

The CLR presents an important opportunity for Scheme. It provides a vast API covering most of the OS services and is an active development platform currently supporting more than two dozen different languages. In this role as a universal binary standard the CLR is even more enticing. By providing access to the CLR, Scheme implementers gain access to libraries written in a number of languages including C++, Lisp, Smalltalk and, ironically, other Schemes.

These facts have not gone unnoticed in other language communities including Haskell, Ruby and Perl which already provide some sort of FFI integration with the CLR [4, 9, 1]. Scheme has some potential advantages in this area however. Its unique syntax definition capabilities can arguably be used to achieve a simpler and more natural result than what is possible in other languages.

5. REFERENCES

[1] ActiveState. PerlNet http://aspn.activestate.com/ASPN/docs/PDK/PerlNET/PerlNET.html

[2] Box, D. and Sells, C. Essential .NET, Volume I: The Common Language Runtime. Addison Wesley, 2002, Ch. 6.

[3] Dybvig, R. Writing Hygienic Macros in Scheme With Syntax-Case, Indiana University Computer Science Department Technical Report #356, 1992

[4] Finne, S. Hugs98 for .NET, http://galois.com/~sof/hugs98.net/

[5] Flatt, M. PLT MzScheme: Language Manual, 2003, Ch. 5

[6] Flatt, M. Inside PLT MzScheme, 2003.

[7] PLT, http:///www.drscheme.org

[8] Richter, J. Applied Microsoft .NET Programming. Microsoft Press, 2002. Ch. 19.

[9] Schroeder B, Pierce J. Ruby/.NET Bridge http://www.saltypickle.com/rubydotnet/

[10] Thai, T. and Lam H. .NET Framework Essentials. O'Reilly, 2001

From Python to PLT Scheme

Philippe Meunier Daniel Silva

College of Computer and Information Science
Northeastern University
Boston, MA 02115
{meunier,dsilva}@ccs.neu.edu

ABSTRACT

This paper describes an experimental embedding of Python into DrScheme. The core of the system is a compiler, which translates Python programs into equivalent MzScheme programs, and a runtime system to model the Python environment. The generated MzScheme code may be evaluated or used by DrScheme tools, giving Python programmers access to the DrScheme development suite while writing in their favorite language, and giving DrScheme programmers access to Python. While the compiler still has limitations and poor performance, its development gives valuable insights into the kind of problems one faces when embedding a real-world language like Python in DrScheme.

1. INTRODUCTION

The Python programming language [13] is a descendant of the ABC programming language, which was a teaching language created by Guido van Rossum in the early 1980s. It includes a sizeable standard library and powerful primitive data types. It has three major interpreters: CPython [14], currently the most widely used interpreter, is implemented in the C language; another Python interpreter, Jython [11], is written in Java; Python has also been ported to .NET [9].

MzScheme [8] is an interpreter for the PLT Scheme programming language [7], which is a dialect of the Scheme language [10]. MzScheme compiles syntactically valid programs into an internal bytecode representation before evaluation. MrEd [6] is a graphical user interface toolkit that extends PLT Scheme and works uniformly across several platforms (Windows, Mac OS X, and the X Window System.) Originally meant for Scheme, DrScheme [5] is an integrated development environment (IDE) based on MzScheme—it is a MrEd application—with support for embedding third-party extensions. DrScheme provides developers with useful and modular development tools, such as syntax or flow analyzers. Because MzScheme's syntax system includes precise source information, any reference by a development tool to such data can be mapped back to a reference to the original program text.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Philippe Meunier and Daniel Silva.

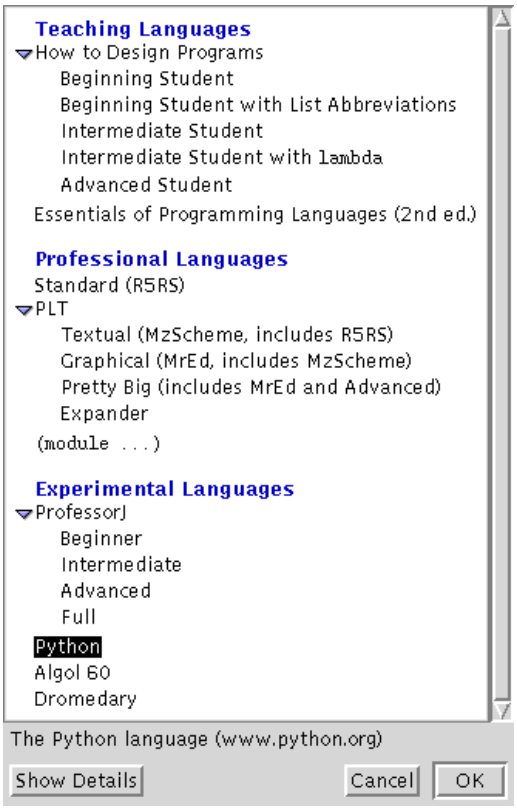


Figure 1: DrScheme language selection menu

DrScheme is thus no longer just a development environment for Scheme. It can now potentially play the role of a program development environment for any language, which users can select from a menu (figure 1). When using any language from within the IDE, the program developer may use DrScheme's development tools, such as Syntax Check, which checks a program's syntax and highlights its bindings (figure 2), or MrFlow, which analyses a program's possible flow of values (MrFlow is still under development though). Also, any new tool added to the DrScheme IDE is supposed to work automatically with all the languages that DrScheme supports (figure 2).

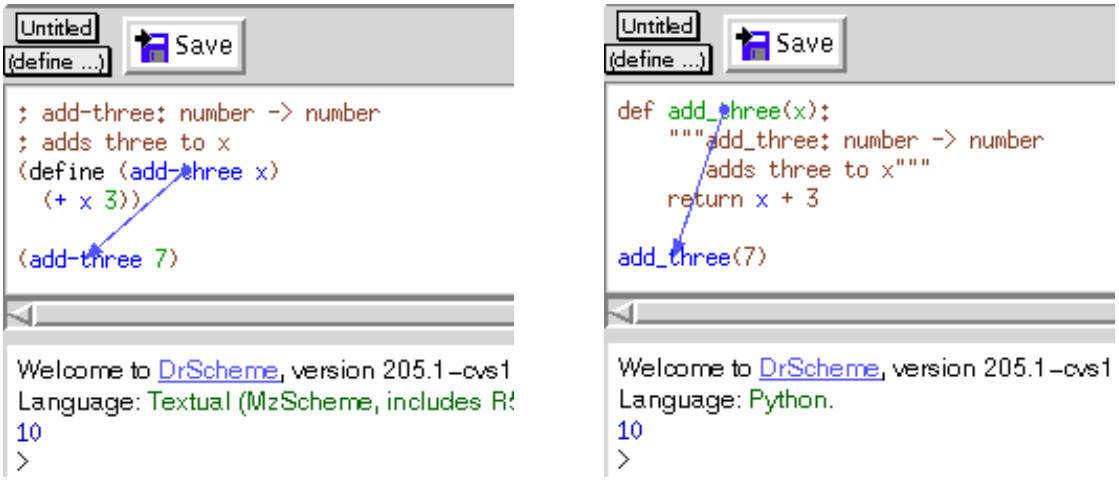


Figure 2: Evaluation and Syntax Check for Scheme and Python

To support a new language, however, DrScheme needs a translator for programs written in that language. In the case of adding Python support to DrScheme, this is the task of the Python-to-Scheme compiler described in this paper. The compiler is packaged as a DrScheme language tool, thus introducing Python as a language in DrScheme's list of choices (figure 1).

The compiler was created as an experiment in porting a language like Python to DrScheme. With Python available as a DrScheme language, Python programmers can use the DrScheme IDE and its accompanying tools to develop Python programs. It also gives Scheme programmers access to the large amount of Python code available on the Internet. The compiler still suffers from several limitations though, primarily relating to the runtime support. The performance of the generated code is also currently poor compared to CPython. While we expect some of the limitations to disappear in the future and the performance to get better as the generated code is optimized, we already consider the experiment to be successful for the insights we have gained about the problem of embedding a real-world language into DrScheme.

Section 2 of this paper presents the overall architecture of the compiler system, including details about code generation and the runtime system. Section 3 describes the current status of the compiler, gives an idea of the current performance of the generated MzScheme code, and evaluates the successfulness of the whole experiment. Section 4 relates other works to this paper. Section 5 lists some of the major parts that still need to be worked on, and we conclude in section 6.

2. ARCHITECTURE

This section describes the architecture of the Python-to-Scheme compiler. The compiler has a conventional structure with three major components: the front-end, which uses a lexical analyzer to read program text and a parser to check the syntax of the tokens produced by the scanner; the back-end, which is a code generator using the parser's output to create MzScheme code; and the runtime system, which provides low-level functions that the generated code makes use of. This section delineates these three components. Section 2.1 describes the scanner and parser; section 2.2, the code generator; and section 2.3, the runtime system.

Note that, even though CPython is based on a virtual machine, we did not consider compiling CPython byte code instead of compiling Python source code. While compiling CPython byte code to Scheme is certainly doable, the semantic mismatch between the stack-based byte code and Scheme is big enough that DrScheme's tools would most likely give poor results on byte code (in addition to the problem of mapping those results for the byte code back into results for Python source code, since, unlike DrScheme, CPython does not preserve in the byte code much information about the source code to byte code transformation).

2.1 Lexical and Syntax Analysis

Python program text is read by the lexical analyzer and transformed into tokens, including special tokens representing indentation changes in the Python source code. From this stream of tokens the parser generates abstract syntax trees (ASTs) in the form of MzScheme objects, with one class for each Python syntactic category. The indentation tokens are used by the parser to determine the extent of code blocks. The list of generated ASTs is then passed on to the code generator.

2.2 Code Generation

The code generator produces Scheme code from a list of ASTs by doing a simple tree traversal and emitting equivalent MzScheme code. The following subsections explain the generation of the MzScheme code for the most important parts of the Python language. They also describe some of the problems we encountered.

2.2.1 Function Definitions

Python functions have a few features not present in Scheme functions. Tuple variables are automatically unpacked, arguments may be specified by keyword instead of position, and those arguments left over (for which no key matches)

are placed in a special dictionary argument. These features are implemented using a combination of compile-time rewriting (e.g. for arguments specified by keywords) and runtime processing (e.g. conversion of leftover arguments into a Python tuple). Default arguments are not yet implemented. Python's `return` statement is emulated using MzScheme escape continuations. For example the following small Python function:

```
def f(x, y, z, *rest, **dict):
    print dict
```

is transformed into the following Scheme definition:

```
(namespace-set-variable-value! 'f
  (procedure->py-function%
   (opt-lambda (dict x y z . rest)
     (let ([rest (list->py-tuple% rest)])
       (call-with-escape-continuation
        (lambda (return10846)
          (py-print #f (list dict))
          py-none))))
   'f (list 'x 'y 'z) null 'rest 'dict))
```

2.2.2 Function Applications

Functions are applied through `py-call`. A function object is passed as the first argument to `py-call`, followed by a list of supplied positional arguments (in the order they were supplied), and a list of supplied keyword arguments (also in order). So, for example, the function call `add_one(2)` becomes:

```
(py-call add_one
         (list (number->py-number% 2))
         null)
```

The `py-call` function extracts from the `add_one` function object a Scheme procedure that simulates the behavior of the Python function when it is applied to its simulated Python arguments by `py-call`.

2.2.3 Class Definitions

In Python classes are also objects. A given class has a unique object representing it and all instances of that class use a reference to that unique class object to describe the class they belong to. The class of class objects (i.e. the type of an object representing a type) is the `type` special object / class. The type of `type` is `type` itself (i.e. `type` is an object whose class is represented by the object itself). With this in mind consider this small Python class, which inherits from two classes `A` and `B` that are not shown here:

```
class C(A, B):
    some_static_field = 7
    another_static_field = 3

    def m(this, x):
        return C.some_static_field + x
```

In this class `C`, three members are defined: the two static fields and the method `m`, which adds the value of the first static field to its argument. This class is converted by the code generator into a static method call to the `__call__` method of the `type` class (which is also a callable object). This call returns a new class object which is then assigned to the variable `C`:

```
(namespace-set-variable-value! 'C
  (python-method-call type '__call__
   (list
    (symbol->py-string% 'C)
    (list->py-tuple% (list A B))
    (list
     (lambda (this-class)
       (list 'some_static_field
             (number->py-number% 7)))
     (lambda (this-class)
       (list
        'another_static_field
        (let-values
            ([(some_static_field)
              (values (python-get-member
                       this-class
                       'some_static_field #f))])
          (number->py-number% 3))))
     (lambda (this-class)
       (list
        'm
        (procedure->py-function%
         (opt-lambda (this x)
           (call/ec
            (lambda (return)
              (return
               (python-method-call
                (python-get-attribute
                 C 'some_static_field)
                '__add__
                (list x)))
              py-none)))
         'm (list 'this 'x) null #f #f)))))))
```

All instances of the class `C` then refer to that new class object to represent the class they belong to.

At class creation time member fields (but not methods) have access to the previously created fields and methods. So for example `some_static_field` must be bound to its value when evaluating the expression used to initialize the field `another_static_field` in the class `C` above. To emulate this the generated Scheme code that initializes a field must always be a function that receives as value for its `this-class` argument the class object currently being created, to allow for the extraction of already created fields and methods from that class object if necessary.

Note that a class's type is different from a class's parent classes. The parents of a class (the objects `A` and `B` representing the parent classes of `C` in the example above) can be accessed through the `__bases__` field of a class. The `__class__` field of an "ordinary" object refers to the object representing that object's class while the `__class__` field of an object representing a class refers to the `type` object (figure 3). This second case includes the `__class__` field of the top `object` class, even though `type` is a subclass of `object`. The class of an object can be changed at runtime by simply assigning a new value to the `__class__` field of that object.

The Python object system also allows fields and methods to be added to an object at runtime. Since classes are themselves objects, fields and methods can be added at runtime

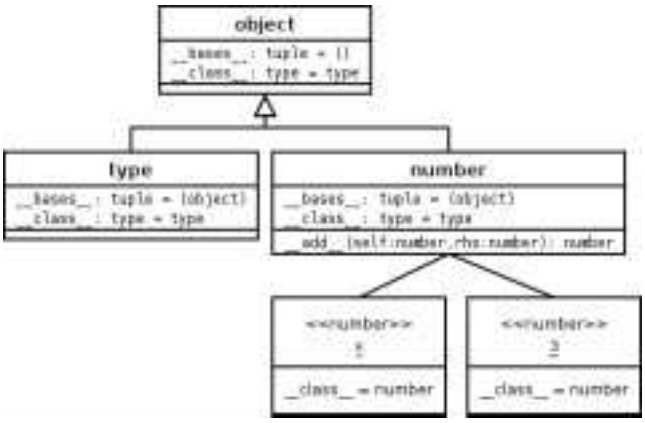


Figure 3: A simple Python class

to a class, which is then reflected in all the existing instances of that class.

Since the MzScheme object system segregates classes and objects, does not allow either to be modified at runtime, and does not support multiple inheritance, the Python object system could not be mapped to the MzScheme one. All Python object are therefore emulated using MzScheme hash tables (which is also what they are internally in CPython).

2.2.4 Variable Assignments

Identifiers are normally bound either at the top level or inside functions. Identifiers from imported modules are bound differently (see section 2.2.5).

Assignments at the top level are translated into `define`s for first assignments or `set!`s for mutative assignments. In the following Python listing, the first line defines `x`, while the second line mutates `x` and defines `y` as the same value `2` (which is only evaluated once).

```
x = 1
x = y = 2
```

Identifiers defined inside functions are bound using `let`. For example, consider the following function that uses a single variable, `x`, defined on the fly.

```
def f():
    x = 1
```

Its body is translated into this Scheme equivalent (omitting the escape continuation code used to handle possible `return` statements):

```
(namespace-set-variable-value! 'f
  (opt-lambda ()
    (let ([x (void)])
      (let ([rhs1718 (number->py-number% 1)])
        (set! x rhs1718))
      py-none)))
```

As a current shortcoming of the compiler, all variables defined throughout the body of a Python function are defined at once in a single `let` at the start of the corresponding Scheme function. To ensure that using a variable before it is defined still results in a runtime error the `let`-bound variables have to be given the value `void`. While this works fine in practice, it does not provide for good error messages though. This will be fixed in the future (see section 5).

When a `global` statement names any variable, the named variable is simply omitted from the Scheme function's initial `let` bindings, thereby allowing assignments to said variable to mutate an identifier existing in an outer scope (if it exists, otherwise a runtime error occurs).

2.2.5 Importing Modules

Unlike MzScheme modules, Python modules allow assignments to identifiers defined in other modules. Python also allows cycles between modules. It was therefore not possible to map Python modules to MzScheme modules. Rather Python modules are emulated using MzScheme namespaces.

In order to import a Python module at runtime—and, in fact, to initialize the environment at startup—the runtime system creates a new MzScheme namespace and populates it with the built-in Python library. The runtime system then compiles the requested module and evaluates it in this new namespace. Finally, new bindings for the necessary values are copied from that namespace into the original namespace of the module importer. For example, when evaluating the statement `import popen from os`, only the binding for `popen` is copied into the original namespace from the new one created to compile the `os` module. A module is always only compiled once, even if it is imported multiple times.

Since `import m` only copies over a reference to module `m` and its namespace, references to values in module `m`, such as `m.x`, are shared between modules importing `m`. However, a statement of the form `from m import x` copies the value of `x` into the current module namespace. There is no sharing of `x` between modules then.

2.3 The Runtime System

The Python runtime system can be divided into two parts: modules that are written in Python and modules that are written in C. The code generation described above can be applied to both user code and the parts of the Python runtime that are written in Python. This means that Python programmers can use these runtime modules as they would normally do. This also means that Scheme programmers have access to the parts of the Python runtime written in Python by simply invoking the compiler on them and evaluating the resulting MzScheme code (although there is currently no simple API provided to do that).

The C-level modules of the Python runtime can be dealt with in several ways. Some of these modules use C macros to abstract the runtime code over the actual internal representation of Python objects. These modules can therefore in principle be directly reused by modifying the appropriate C macros to work on MzScheme values instead of Python objects. The use of C macros is not systematic throughout the Python runtime code though, so some changes to the code are required to make it completely abstract and there does not seem to be any simple automated way to do this. As an experiment the Python String class code and macros were modified in this manner and the class is now usable by the DrScheme Python programmer.

For the Python modules written in C that are poorly, or not at all, abstracted over the representation of Python object, the most elegant solution would be to convince the CPython developers to rewrite these core modules in a more abstract way using C macros, thereby allowing the two systems to share that runtime code. We do not expect this to happen in the foreseeable future though, so one alternative solution is to replace these C modules with equivalent MzScheme code. Calls to Python runtime functions can be transformed by the code generator into calls to MzScheme functions when the Python functions have direct MzScheme equivalents (e.g. `printf`). Python functions that do not have any direct MzScheme equivalent must be rewritten from scratch, though this brings up the problem of maintaining consistency with the CPython runtime as it changes. We are currently examining the Python C code to determine how much of it can be reused and how much of it has to be replaced. Another possible solution is to use an automated tool like SWIG [3] to transform the Python C modules into MzScheme extensions. The code generator can then replace calls to the original C modules by MzScheme function calls to the SWIG-generated interface. This approach is also under investigation.

Note that there is currently no way for the Python programmer using DrScheme to access the underlying MzScheme runtime. Giving such access is easy to do through the use of a Python module naming convention that can be treated as a special case by the code generator (e.g. `import mzscheme` or `import ... from mzscheme`).

3. STATUS AND EVALUATION

Most of the Python language has been implemented, with the exception of the `yield` and `exec` statements, and of default function parameters (as explained in section 2.2.1). The Python `eval` function has not been implemented yet either but since `import` is implemented and since it evaluates entire Python files, the necessary machinery to implement both `exec` and `eval` is already available. There is no plan to support Unicode strings, at least as long as MzScheme itself does not support them. There is also currently no support for documentation strings. As described in section 2.3 access to the parts of the Python runtime system written in C is still a problem.

Because Python features like modules or objects have very dynamic behaviors and therefore must be emulated using MzScheme namespaces and hash tables (respectively), the code generated by our system is in general significantly bigger than the original Python code. See for example the simple Python class from section 2.2.3 that expands into about 30 lines of MzScheme code. In general a growth factor of about three in the number of lines of code can be expected. The generated code also involves a large number of calls to internal runtime functions to do anything from continually converting MzScheme values into Python values and back (or more precisely into the internal representation of Python values our system is using and back) to simulating a call to the `__call__` method of the `type` class object. Finally, each module variable, class field or method access potentially involves multiple namespace or hashtable lookups done at the Scheme level. As a result the performance of the resulting code is poor compared to the performance of the original Python code running on CPython. While no systematic performance measurement has been made yet, anecdotal evidence on a few test programs shows a slowdown by around three orders of magnitude.

Using DrScheme tools on Python programs has given mixed results. Syntax Check, which checks a program's syntax and highlights its bindings using arrows, has been successfully used on Python programs without requiring any change to the tool's code (figure 2). Some features of the Python language make Syntax Check slightly less useful for Python programs than for Scheme programs though. For example, since an object can change class at runtime, it is not possible to relate a given method call to a specific method definition using just a simple syntactic analysis of the program. This is a limitation inherent to the Python language though, not to Syntax Check.

A tool like MrFlow, which statically analyzes a program to predict its possible runtime flow of values, could potentially be able to relate a given method call to a given method definition. While MrFlow can already be used on Python programs without any change, it does not currently compute any meaningful information: MrFlow does not know yet how to analyze several of the MzScheme features used in the generated code (e.g. namespaces). Even once this problem is solved, MrFlow will probably still compute poor results. Since all Python classes and object are emulated using MzScheme hash tables, and since value flow analyses are unable to differentiate between runtime hash table keys, MrFlow will compute extremely conservative results for all the object oriented aspects of a Python program. In general there is probably no easy way to statically and efficiently analyze the generated code. In fact there is probably no way to do good value-flow analysis of Python programs at all given Python's extremely dynamic notion of objects and modules.

Another DrScheme tool, the Stepper, does not currently work with Python programs. The Stepper allows a programmer to run a program interactively step by step. To work with Python the Stepper would need to have access to a decompiler, a program capable of transforming a generated but reduced MzScheme program back into an equivalent Python program. Creating such an decompiler is a non-trivial task given the complexity of the code generated by the compiler.

Due to the difficulties encountered with the Python runtime and due to the current poor performance of the code generated, the compiler should be considered to be still at an experimental stage. The fact that most of the Python language has been implemented and that a DrScheme tool like Syntax Check can be used on Python programs without any change is encouraging though. The experience that has been gained in porting a real-world language to DrScheme is also valuable. We therefore consider the experiment to be successful, even if a lot of work still remains to be done.

4. RELATED WORK

Over the past years there have been several discussions [1, 2] on Guile related mailing lists about creating a Python to Guile translator. A web site [4] for such a project even exists, but does not contain any software. Richard Stallman indicated [12] that a person has been working on "finish-

ing up a translator from Python to Scheme" but no other information on that project could be found.

Jython, built on top of the Java Virtual Machine, is another implementation of the Python language. The implementation is mature and gives users access to the huge Java runtime. All the Python modules implemented in C in CPython were simply re-implemented in Java. Maintaining the CPython and Jython runtimes synchronous requires constant work though.

Python for .NET is an exploratory implementation of the Python language for the .NET framework and has therefore severe limitations (e.g. no multiple inheritance). Like Jython it gives access to the underlying runtime system and libraries. Only a handful of modules from the Python runtime have been implemented. Among those, the ones written in Python became accessible to the user after being modified to fit within the more limited Python language implemented by the interpreter. A few modules originally written in C in CPython were re-implemented using the C# language.

Compilers for other languages beside Python are being developed for DrScheme. Matthew Flatt developed an implementation of the Algol60 language as a proof of concept. David Goldberg is currently working on a compiler for the OCaml language called Dromedary, and Kathy Gray is working on a DrScheme embedding of Java called ProfessorJ.

5. FUTURE WORK

In its present state the biggest limitation of the compiler is the lack of access to the C-level Python runtime. As such we are currently focusing most of our development efforts in that area, investigating several strategies to overcome this problem (see section 2.3).

While the performance of the generated code is poor, no attempt has yet been made at profiling it. The performance will be better once the code generator has been modified to create more optimized code, although it is unclear to us at this stage how much improvement can be expected in this regard. The need to simulate some of the main Python features (e.g. the object and module systems) and the large number of runtime function calls and lookups involved means than the generated code will probably never have a performance level on par with the CPython system although an acceptable level should be within reach.

As described in section 3, a few parts of the Python language remain to be implemented. We do not anticipate any problem with these. There is also a general need for better error messages and a more complete test suite.

6. CONCLUSION

A new implementation of the Python language is now available, based on the MzScheme interpreter and the DrScheme IDE. While most of the core language has been implemented a lot of work remains to be done on the implementation of the Python runtime and on improving the performance. Despite this Python developers can already benefit from some of DrScheme's development tools to write Python code, and Scheme programmers start now to have access to the large number of existing Python libraries.

7. ACKNOWLEDGMENT

Many thanks to Scott Owens for developing the DrScheme parser tools and creating the first version of the MzScheme lexer and parser grammar for the Python language.

8. REFERENCES

[1] Guile archive. `http://www.cygwin.com/ml/guile/2000-01/threads.html#00382`, January 2000.

[2] Guile archive. `http://sources.redhat.com/ml/guile/2000-07/threads.html#00072`, July 2000.

[3] David M. Beazley. SWIG Users Manual. `www.swig.org`, June 1997.

[4] Thomas Bushnell. `http://savannah.gnu.org/projects/gpc/`.

[5] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. `Programming Languages: Implementations, Logics, and Programs`, 1292:369–388, September 1997.

[6] Matthew Flatt. `PLT MrEd: Graphical Toolbox Manual` PLT, December 2002.

[7] Matthew Flatt. `PLT MzScheme: Language Manual` PLT, December 2002.

[8] Matthew Flatt. The MzScheme interpreter. `http://www.plt-scheme.org/`, December 2002. Version 203.

[9] Mark Hammond. Python for .NET: Lessons learned. ActiveState Tool Corporation, November 2000.

[10] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. `ACM SIGPLAN Notices`, 33(9):26–76, 1998.

[11] Samuele Pedroni and Noel Rappin. `Jython Essentials` O'Reilly, March 2002.

[12] Richard Stallman. Invited Talk: My Lisp Experiences and the Development of GNU Emacs. In `Proc. International Lisp Conference`, October 2002.

[13] Guido van Rossum. `Python R eference Manual` PythonLabs, February 2003.

[14] Guido van Rossum. The Python interpreter. PythonLabs, February 2003. Version 2.3a2.

How to Add Threads to a Sequential Language Without Getting Tangled Up

Martin Gasbichler Eric Knauel
Institut für Informatik
Universität Tübingen
{gasbichl,knauel}@informatik.uni-tuebingen.de

Michael Sperber*
sperber@deinprogramm.de

Richard A. Kelsey
Ember Corporation
kelsey@s48.org

Abstract

It is possible to integrate Scheme-style first-class continuations and threads in a systematic way. We expose the design choices, discuss their consequences, and present semantical frameworks that specify the behavior of Scheme programs in the presence of threads. While the issues concerning the addition of threads to Scheme-like languages are not new, many questions have remained open. Among the pertinent issues are the exact relationship between continuations and the `call-with-current-continuation` primitive, the interaction between threads, first-class continuations, and `dynamic-wind`, the semantics of dynamic binding in the presence of threads, and the semantics of thread-local store. Clarifying these issues is important because the design decisions related to them have profound effects on the programmer's ability to write modular abstractions.

1 What's in a Continuation?

Scheme [21] was one the first languages to endorse `call-with-current-continuation` as a primitive. `Call-with-current-continuation` (or `call/cc`, for short) is an essential ingredient in the implementation of a wide range of useful abstractions, among them non-local control flow, exception systems, coroutines, non-deterministic computation, and Web programming session management. So much is often repeated, non-controversial and clear.

Nowadays, even the name `call-with-current-continuation` is confusing. It suggests erroneously that `call/cc` applies its argument to a reified version of the current continuation—the meta-level object the underlying machine uses to remember what should happen with the value of the expression currently being evaluated.

*For parts of this work, Michael Sperber was supported by the Institut für Informatik at Universität Tübingen.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.
Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA.
Copyright 2003 Martin Gasbichler, Eric Knauel, Michael Sperber and Richard Kelsey.

The denotational semantics presented in R^{5}RS [21] supports this impression. Here is a slightly simplified version:

$$\mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad \texttt{[call-with-current-continuation]}$$
$$\mathit{cwcc} = \mathit{onearg}\,(\lambda\varepsilon\kappa\,.\,\mathit{applicate}\,\varepsilon\,((\lambda\varepsilon^*\kappa'\,.\,\kappa\varepsilon^*)\ \text{in}\ \mathtt{E})\,\kappa)$$

The reified value passed to the argument of *cwcc* is the function $\lambda\varepsilon^*\kappa'\,.\,\kappa\varepsilon^*$—essentially an eta-expanded version of κ, the current continuation as handled by the semantics. Calling this function merely re-installs or *reflects* κ as the current continuation. With this definition, the distinction between the *escape procedure*—the procedure passed to `call/cc`'s argument and the actual meta-level continuation is largely academic.

Unfortunately, the semantics for `call/cc` given in R^{5}RS is not correct, as noted in a "Clarifications and corrections" appendix to the published version: an R^{5}RS-compliant `call/cc` must also execute thunks along the branches of the control tree as introduced by the `dynamic-wind` primitive [18] added to Scheme in R^{5}RS. Even in pre-R^{5}RS Scheme, the escape procedure would typically re-install previously captured values for the current input and output ports. Thus, the escape procedure created by `call/cc` performs actions *in addition* to installing a captured continuation. Hence, the name `call-with-current-continuation` is misleading.

`Dynamic-wind` allows enhancing and constraining first-class continuations: (`dynamic-wind` *before thunk after*) calls *thunk* (a procedure of no parameters), ensuring that *before* (also a thunk) is always called before the program enters the application of *thunk*, and that *after* is called after the program has left it. Therefore, escape procedures created by `call/cc` must also call the *after* and *before* thunks along the paths leading from the current node in the control tree to the target tree. This creates a significant distinction between an escape procedure and its underlying continuation.

This distinction has created considerable confusion: Specifically, continuations are suitable abstractions for building thread systems [37], and this suggests that escape procedures are, too. However, a thread system based on R^{5}RS `call/cc` will run *before* and *after* thunks introduced by `dynamic-wind` upon every context switch, which leads to semantic and pragmatic problems in addition to the common conceptual misunderstandings noted by Shivers [32]. Moreover, other common abstractions, such as dynamic binding and thread-local storage, interact in sometimes surprising ways with threads and first-class continuations, depending on their exact semantics in a given system.

Thus, the integration of first-class continuations with `dynamic-wind`, concurrency and parallelism, along with associated functionality such as dynamic binding and thread-local storage form a puzzle: Most of the pieces have long been on the table, but there is little published documentation on how all of them fit together in a systematic way, which often causes confusion for users and implementors alike. With this paper, we try to make the pieces fit, and close some of the remaining gaps.

Here are the contributions of our work:

- We discuss some of the pertinent semantic properties of `dynamic-wind`, specifically as they relate to the implementation of dynamic binding.
- We discuss design issues for thread systems in Scheme-like languages, and how different design choices affect program modularity.
- We present a systematic treatment of two abstractions for thread-aware programming: `thread-wind` extends the context switch operation, and thread-local storage implements extensible processor state.
- We present a denotational semantics of R^{5}RS `call/cc` and `dynamic-wind`.
- We clarify the relationship between threads and `call/cc`/`dynamic-wind` by presenting an transition semantics based on the CEK machine [6] equivalent to the denotational semantics, and extending this semantics by simple models for threads and multiprocessing.

Overview: Section 2 gives an account of `call/cc` as present in (sequential) Scheme, and its interaction with `dynamic-wind`. Section 3 lists some specific design issues pertinent to the addition of threads to Scheme and describes their impact on the ability to write modular programs. More issues arise during implementation; Section 4 discusses these. Section 5 describes facilities for thread-aware programming. Section 6 presents semantic characterizations of Scheme with `dynamic-bind` and threads. Related work is discussed in Section 7; Section 8 concludes.

2 `Call/cc` As We Know It

In this section, we give an informal overview of the behavior of the R^{5}RS Scheme version of `call/cc`. Specifically, we discuss the interaction between `call/cc` and the current dynamic environment implicit in R^{5}RS, and the interaction between `call/cc` and `dynamic-wind`. We also explain how these interactions affect possible implementations of an extensible dynamic environment.

2.1 The current dynamic environment

R^{5}RS [21] implies the presence of a *current dynamic environment* that contains bindings for the current input and output ports. Scheme's I/O procedures default to these ports when they are not supplied explicitly as arguments. Also, the program can retrieve the values of the bindings via the `current-input-port` and `current-output-port` procedures. "Dynamic" in this context means that the values for the program behave as if the current dynamic environment were implicitly passed as an argument with each procedure application. In this interpretation, `with-input-from-file` and `with-output-to-file` each call its argument with a newly created dynamic environment containing a new binding, and `current-{input,output}-port` retrieve

```
(define *dynamic-env* (lambda (v) (cdr v)))

(define (make-fluid default) (cons 'fluid default))

(define (fluid-ref fluid) (*dynamic-env* fluid))

(define (shadow env var val)
  (lambda (v)
    (if (eq? v var)
        val
        (env var))))

(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (set! *dynamic-env* new-env)
    (let ((val (thunk)))
      (set! *dynamic-env* old-env)
      val)))
```

Figure 1. Dynamic binding via dynamic assignment

the values introduced by the most recent, still active application of these procedures. The interpretation of the current dynamic environment as an implicit argument means that dynamic environments are effectively associated with continuations. Specifically, reflecting a previously reified continuation also means returning to the dynamic environment which was current at the time of the reification.[1]

It is often useful to be able to introduce new dynamic bindings [24, 16] in addition to `current-{input,output}-port`, for example to implement exception handling. However, as the dynamic environment is implicit (and not reifiable), a program cannot extend it. Fortunately, it is possible to simulate extending the dynamic environment with first-class procedures by keeping the current dynamic environment in a global variable, and simply save and restore it for new bindings—a technique known as *dynamic assignment* [11].

Figure 1 shows naive code for dynamic assignment. `Make-fluid` creates a fluid represented as a pair consisting of the symbol `fluid` as its `car` and the default value as its `cdr`. `*Dynamic-env*` holds the current dynamic environment, represented as a procedure mapping a fluid to its value. The initial function in `*dynamic-env*` extracts the default value of a fluid. `Shadow` makes a new dynamic environment from an old one, shadowing one binding with a new one. `Bind-fluid` remembers the old value of `*dynamic-env*`, sets it to a new one created by `shadow`, calls `thunk`, and restores the old value. (The code ignores the issue of multiple return values for simplicity.) The `fluid-ref` procedure looks up a fluid binding in the dynamic environment, returning its value.

Unfortunately, `bind-fluid` does not implement the implicit-argument semantics in the presence of `call/cc`: it is possible for the `thunk` argument to `bind-fluid` to reflect a previously reified continuation which will then inherit the current dynamic environment, rather than the dynamic environment current at the time of reification. For implementing the implicit-argument semantics, it is necessary to capture the current value of `*dynamic-env*` at the time of reification, and re-set it to that value upon reflection.

[1] Note that this behavior is not mandated by R^{5}RS. However, existing Scheme code often assumes it [23].

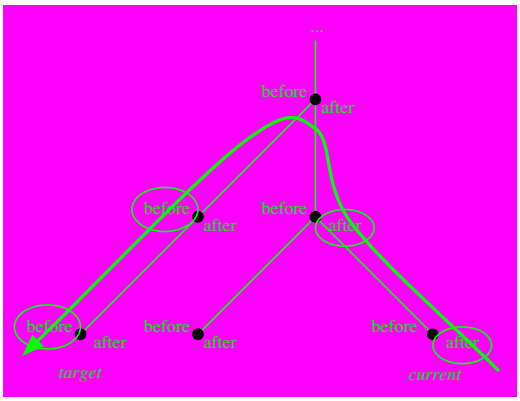


Figure 2. Control tree and `dynamic-wind`

2.2 `Dynamic-wind`

While the naive implementation of dynamic assignment does not have the desirable semantics, it is possible to implement a version that does, via the Scheme primitive `dynamic-wind`. (Very) roughly, (`dynamic-wind` *before thunk after*) ensures that *before* is called before every control transfer into the application of `thunk`, and *after* is called after every control transfer out of it. Here is a new version of `bind-fluids` that utilizes `dynamic-wind` to get the correct semantics:

```
(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (dynamic-wind
       (lambda () (set! *dynamic-env* new-env))
       thunk
       (lambda () (set! *dynamic-env* old-env)))))
```

The behavior of `dynamic-wind` is based on the intuition that the continuations active in a program which uses `call/cc` form a tree data structure called the *control tree* [18]: each continuation corresponds to a singly-linked list of frames, and the continuations reified by a program may share frames with each other and/or with the current continuation. Reflecting a previously reified continuation means making a different node of the tree the current continuation. A Scheme program handles the current control tree node in much the same way as the dynamic environment. Together, they constitute the *dynamic context*.

Conceptually, (`dynamic-wind` *before thunk after*) annotates the continuation of the call to *thunk* with *before* and *after*. Calling an escape procedure means travelling from the current node in the control tree to the node associated with to the previously reified continuation. This means ascending from the current node to the nearest common ancestor of the two nodes, calling the *after* thunks along the way, and then descending down to the target node, calling the *before* thunks. Figure 2 shows such a path in the control tree.

Using `dynamic-wind` for implementing dynamic binding assures that part of the global state—the value of `*dynamic-env*`, in this case—is set up to allow the continuation to run correctly. This works well for dynamic binding, as changes to `*dynamic-env*` are always easily reversible. However, in some situations a continuation might not be able to execute correctly because global state has changed in an irreversible way. Figure 3 shows a typical code fragment which employs `dynamic-wind` to ensure that the program will close an input port immediately after a set of port operations

```
(let ((port (open-input-file file-name)))
  (dynamic-wind
   (lambda ()
     (if (not port)
         (error "internal error")))
   (lambda () ⟨read from port⟩)
   (lambda ()
     (close-input-port port)
     (set! port #f))))
```

Figure 3. Restricting the use of escape procedures

has completed (in the *after* thunk) as well as preventing the program from inadvertently entering the code that performs file I/O after the close has happened. Moreover, the *before* thunk prevents the port access code from being re-entered because the port operations are likely to have caused irreversible state changes.[2] Thus, three main uses for `dynamic-wind` emerge [18]:

1. extending the dynamic context associated with continuations (as in `bind-fluid`)
2. releasing resources used by a region of code after that code has completed (as in Figure 3)
3. preventing the reification of a continuation because its dynamic context cannot be recreated (as in Figure 3)

Item #2 is akin to the default or `finally` clauses of exception handling systems or to the `unwind-protect` facilities in some languages. The unlimited extent of escape procedures created by `call/cc` makes the more general `dynamic-wind` necessary.

The presence of `dynamic-wind` requires a more careful handling of terminology when it comes to continuations: We call the process of turning the meta-level continuation into an object-level value *reification*, and the reverse—re-installing a previously reified continuation—*reflection*. The process of creating an escape procedure (by `call/cc`) is a *capture*; this includes reifying the current continuation. Conversely, *invoking* the escape procedure travels to the target point in control space, installs the dynamic environment, and then reflects the continuation.

3 Design Requirements for Thread Systems

In this section, we consider some of the design issues that arise when adding threads to a higher-order language. We assume that the thread system features a `spawn` operation. `Spawn` starts a new thread and calls *thunk* (a thunk) in that new thread. The thread terminates once *thunk* returns:

(`spawn` *thunk*) procedure

The presence of `spawn` in a language with `call/cc`, `dynamic-wind`, and dynamic binding exposes a number of language design choices, as these features interact in potentially subtle ways. Specifically, the ability to migrate continuations between threads, and the interaction between dynamic binding and threads fundamentally affect the ability to write modular programs.

3.1 Migrating continuations

A Scheme program can invoke an escape procedure in a thread different from the one where it was captured. Notably, this scenario

[2] Even though it might be possible to redo changes on a file port, this is usually impossible with, say, a network connection.

occurs in multithreaded web servers which use `call/cc` to capture the rest of a partially completed interaction between the server and a client: typically, the server will create a new thread for each new request and therefore must be able to invoke the escape procedure that was captured in the thread which handled the connection belonging to the previous step in the interaction [28].

In MrEd, the Scheme platform on which PLT's web server is based, continuations are "local to a thread"—only the thread that created an escape procedure can invoke it, forcing the web server to associate a fixed thread with a session [14].[3] While this may seem like a technical restriction with a purely technical solution, this scenario exposes serious general modularity issues: Modules may communicate escape procedures, and tying an escape procedure to a thread restricts the implementation choices for a client which needs to invoke an escape procedure created by another module. If the escape procedure is thread-local, the client cannot even tell if invoking it might make the program fail; all it knows is that the invocation will *definitely* fail if performed in a freshly created thread.

Once continuations are allowed to migrate between threads, additional questions arise. In particular, the use of certain abstractions might make the continuation sensitive to migration, which is usually not what the programmer intended.

3.2 Dynamic binding and the thread system

Consider the following program fragment:

```
(define f (make-fluid 'foo))

(bind-fluid f 'bar
  (spawn
   (lambda ()
     (display (fluid-ref f)))))
```

Should the program print `foo` or should it print `bar`? This is a well-known design issue with thread systems [13]. The general question is this: Should a newly spawned thread inherit the dynamic environment from the original thread—or, more precisely, from the continuation of the call to `spawn`—or should it start with an empty dynamic environment, assuming the default values for all fluids?[4]

For at least two dynamic entities, inheritance does not make sense: the current control tree node and, if present, the current exception handler, as they both conceptually reach back into the part of the control tree belonging to the original thread. Thus, it is unclear what should happen if the new thread ever tries to travel back to that part of the tree. (For `dynamic-wind`, we discuss another closely related issue in Section 4.1.) Instead, a newly spawned thread must start with a fresh current exception handler and an empty control tree.

For all other dynamic bindings, it is unclear whether a single inheritance strategy will satisfy the needs of all programs. For many entities typically held in fluids, it makes sense for a new thread to inherit dynamic bindings from the thread which spawned it:

- Scsh [31], tries to maintain an analogy between threads and Unix processes, and keeps Unix process resources in fluids [13]. In Scsh, a special `fork-thread` operation acts like `spawn`, but has the new thread inherit the values of the process resources from the original thread.
- MzScheme [9] provides abstractions for running a Scheme program in a protected environment, thus providing operating-system-like capabilities [10]. Some of the entities controlling the encapsulation of such programs are held in fluids (called *parameters* in MzScheme), such as the current custodian that controls resource allocation and destruction. Children of an encapsulated thread inherit the custodian of the parent so that shutting down the custodian will kill the encapsulated thread along with all of its children.
- Generally, programmers might expect `(spawn` f`)` to behave as similarly as possible to `(`f`)`. This is especially likely if the programmer uses threads to exploit parallelism, in a similar way to using futures [15], and thus merely wants to offload parts of the computation to a different processor.

```
(define (current-dynamic-context)
  (let ((pair (call-with-current-continuation
               (lambda (c) (cons #f c)))))
    (if (car pair)
        (call-with-values (car pair) (cdr pair))
        (cdr pair))))

(define (with-dynamic-context context thunk)
  (call-with-current-continuation
   (lambda (done)
     (context (cons thunk done)))))

(define (spoon thunk)
  (let ((context (current-dynamic-context)))
    (spawn
      (lambda ()
        (with-dynamic-context context thunk)))))
```

Figure 4. Reifying and reflecting the dynamic context

3.3 Dynamic binding and modularity

The issue of fluid inheritance is most pertinent when a program module keeps mutable data in fluids. Specifically, consider the following scenario: Program module A creates and uses fluids holding mutable state. The fluids might be exported directly, or module A might provide `with-`f abstractions roughly like the following:

```
(define f (make-fluid default))

(define (with-f value thunk)
  (bind-fluid f (... value ...) thunk))
```

A client of module A might want to create multiple threads, and use the abstractions of module A from several of them. Generally, the client might need to control the sharing of state held in f for each new thread it creates in the following ways:

1. getting A's default dynamic bindings,
2. creating a new binding for A by using `with-`f in the new thread, or
3. inheriting the current thread's dynamic environment.

If each thread starts up with a fresh dynamic environment, this degree of control is available:

1. Starting a new thread with a fresh dynamic environment means that it will get default bindings for all fluids.

[3] This restriction will be lifted in a future version of MrEd.

[4] The issue becomes more subtle with SRFI-18-like thread systems [5] with separate `make-thread` and `thread-start!` operations. Whose dynamic environment should the new thread inherit?

2. Explicitly creating new bindings is possible via `with-`f.
3. It is still possible to implement a variant of `spawn` that does cause the new thread to inherit the dynamic environment from the thread which created it.

Figure 4 shows how to achieve the last of these: As `call/cc` captures the dynamic context, it is possible to reify and reflect it, along with the dynamic environment, through escape procedures. `Current-dynamic-context` uses `call/cc` to create an escape procedure associated with the current dynamic context, and packages it up as the `cdr` of a pair. The `car` of that pair is used to distinguish between a normal return from `call/cc` and one from `with-dynamic-context` which runs a thunk with the original continuation—and, hence, the original dynamic context—in place and restores its own continuation after the thunk has finished. With the help of these two abstractions, `spoon` (for a "fluid-preserving fork operation for threads," a term coined by Alan Bawden) starts a new thread which inherits the current dynamic environment.[5]

Note that `spoon` causes the new thread to inherit the *entire* dynamic context, including the current control tree node, and the current exception handler (if the Scheme system supports exception handling in the style of ML.) This can lead to further complications [1]. Also, inheritance is not the only possible solution to the security requirements of MrEd: Thread systems based on nested engines [2] such as that of Scheme 48 allow defining custom schedulers. Here, a scheduler has full control over the initial dynamic environment of all threads spawned with it.

4 Implementing Concurrency

The previous section has already stated some of the design requirements and choices for an implementation of threads in a language with first-class continuations. Additional issues emerge when actually implementing threads in the presence of `call/cc` and `dynamic-wind`. In particular, many presentations of thread systems build threads *on top* of the language, using `call/cc` to implement the context switch operation. However, this choice incurs undesirable complications (especially in the presence of multiprocessing) when compared to the alternative—implementing threads primitively and building the sequential language on top.

4.1 `Dynamic-wind` vs. the context switch

The presence of `dynamic-wind` makes `call/cc` less suitable for implementing context-switch-like abstractions like coroutines or thread systems: Uses of `dynamic-wind` may impose restrictions on the use of the escape procedures incompatible with context switching.[6] Consider the code from Figure 3. This code should continue to work correctly if run under a Scheme system with threads—say, in thread X. However, if the context switch operation of thread system is implemented using ordinary `call/cc`, each context switch out of thread X means ascending up the control tree to the scheduler (whose continuation frames constitute the shared part of tree)—executing all *after* thunks of all `dynamic-wind` operations active within the current thread. The next context switch back into thread X will then run the *before* thunks, which in this case will make the program fail. Naturally, this is unacceptable.

Moreover, if every context switch would run `dynamic-wind` *before* and *after* thunks, the program would expose the difference between a virtualized thread system running on a uniprocessor and a multiprocessor where multiple threads can be active without any context switch: If each thread ran on a different processor, no continuations would ever be captured or invoked for a context switch, so a context switch would never cause `dynamic-wind` thunks to run.

Thus, building a thread system on top of R^{5}RS `call/cc` leads to complications and invalidates common uses of `dynamic-wind`. (Similar complications occur in the presence of ML-style exception handling [1].) Hence, a more reasonable approach for implementations is to build threads natively into the system, and build `call/cc` and `dynamic-wind` on top of it. In this scenario, each newly spawned thread starts with an empty dynamic context.

4.2 Dynamic binding vs. threads

In the presence of threads, the implementation of dynamic binding that keeps the current dynamic environment in a global variable no longer works: all threads share the global variable, and, consequently, any application of `bind-fluid` is visible in other threads, violating the intended semantics. Therefore, it is necessary to associate each thread with its own dynamic environment. Here are some possible implementation strategies:

1. pass the dynamic environment around on procedure calls as an implicit argument
2. keep looking for dynamic bindings in the `*dynamic-env*` global variable, and change the value of this variable upon every context switch, always setting it to the dynamic environment associated with the current thread
3. like #2, but keep the dynamic environment in the thread data structure, and always access that instead of a global variable

#1 incurs overhead for every single procedure call; considering that access and binding of fluid variables is relatively rare, this is an excessive cost rarely taken by actual implementations. #2 is incompatible with multiprocessing, as multiple threads can access fluid variables without intervening context switches. #3 is viable.

All of these strategies require what is known as "deep binding" in the Lisp community—`fluid-ref` always looks up the current value of a fluid variable in a table, and only reverts to the top-level value stored in the fluid itself when the table does not contain a binding. Many Lisp implementations have traditionally favored "shallow binding" that manages dynamic bindings by mutating the fluid objects themselves. With shallow binding, access to a fluid variable is simply dereferencing the fluid object; no table searching is necessary. However, this technique is also fundamentally incompatible with multiprocessing because it mutates global state.

4.3 Virtual vs. physical processors

The previous two sections have shown that a multiprocessor thread system can potentially expose differences in implementation strategies for dynamic binding, as well as different ways of dealing with

[5] The same trick is applicable to promises which exhibit the same issues, and which also do not capture the dynamic context: the fluid-preserving versions of `delay` and `force` would be called `freeze` and `thaw`.

[6] Note that this is an inherent issue with the generality of `call/cc`: `Call/cc` allows capturing contexts which simply are not restorable because they require access to non-restorable resources. Providing a version of `call/cc` which does not capture the dynamic context would violate the invariants guaranteed by `dynamic-wind` and break most code which uses it.

dynamic-wind. These differences all concern the notion of "what a thread is"—specifically, if a thread encompasses the dynamic context, or if it is an exterior, global entity.

A useful analogy is viewing a thread as a virtual processor [32] running on a physical processor. In this view, the dynamic context and the dynamic environment are akin to processor registers. In a multiprocessor implementation of threads, each physical processor indeed must keep those values in locations separate from that of the other processors. Each of these processors can then run multiple threads, swapping the values of these registers on each context switch. (This corresponds to Shivers's notion of "continuation = abstraction of processor state" as the entity being swapped upon a context switch [32].) In this model, a thread accessing these registers cannot distinguish whether it is running in a uniprocessor or a multiprocessor system.

5 Thread-Aware Programming

The previous two sections have focused on protecting sequential programs from the adverse effects resulting from the presence of threads, and on decoupling previously present sequential abstractions such as dynamic-wind and dynamic binding from the thread system as far as possible. However, the implementations of low-level abstractions occasionally benefit from access to the guts of the thread system. Two abstractions provide this access in a systematic way: the thread-wind operation allows running code local to a thread upon context-switch operations, and *thread-local cells* are an abstraction for managing thread-local storage. However, the use of these facilities requires great care to avoid unexpected pitfalls.

5.1 Extending the context switch operation

Accessing state like the dynamic-wind context or the dynamic environment through processor registers is convenient and fast. However, as the scheduler needs to swap the values of these registers on each context switch, they are not easily extensible: each new register requires an addition to the context-switch operation. Also, it is occasionally desirable that a thread is able to specify code to be run whenever control enters or exits that thread, thus making the context switch operation extensible. (Originally, dynamic-wind had precisely that purpose, but, as pointed out in Section 4.1, this is not reasonable in light of current usage of dynamic-wind.) Therefore, we propose a new primitive:

(thread-wind *before thunk after*) procedure

In a program with only a single thread, thread-wind acts exactly like dynamic-wind: *before*, *thunk*, and *after* are thunks; they run in sequence, and the thread-wind application returns whatever *thunk* returns. Moreover, *before* gets run upon each control transfer into the application *thunk*, and *after* gets run after each transfer out of it. Unlike with dynamic-wind, however, during the dynamic extent of the call to *thunk*, every context switch out of the thread runs the *after* thunk, and every context switch back in runs the *before* thunk.

Thread-wind is a low-level primitive; its primary intended purpose is to control parts of the processor state not managed by the underlying, primitive thread system. For example, in a uniprocessor setting, it is possible to continue treating the variable *dynamic-env* as a sort of register, and implement bind-fluid correctly by using thread-wind instead of dynamic-wind:

```
(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (thread-wind
     (lambda () (set! *dynamic-env* new-env))
     thunk
     (lambda () (set! *dynamic-env* old-env)))))
```

The semantics of thread-wind extends smoothly to the escape procedure migration scenario: in this case, before the program installs the new continuation, it runs the active thread-wind *after* thunks of the current thread, and the active *before* thunks of the continuation being reflected.

Ideally, the *before* and *after* thunks are transparent to the running thread in the sense that running *after* invalidates whatever state changes *before* has performed. Still, it is possible to use thread-wind to set up more intrusive code to be run on context switches, such as profiling, debugging, or benchmarking.

5.2 Thread-local storage

The version of bind-fluid using thread-wind still is not correct in the presence of multiprocessing, as all processors share the value of *dynamic-env*. For correctly implementing dynamic binding, another conceptual abstraction is needed: *thread-local storage*. Thread-local storage is available through *thread-local cells* or *thread cells* for short. Here is the interface to thread-local cells:

(make-thread-cell *default*) procedure
(thread-cell-ref *thread-cell*) procedure
(thread-cell-set! *thread-cell value*) procedure

Make-thread-cell creates a reference to a thread cell with default value *default*, thread-cell-ref fetches its current value, and thread-cell-set! sets it. Any mutations of a thread cell are only visible in the thread which performs them. A thread cell acts like a table associating each thread with a value which defaults to *default*; thread-cell-ref accesses the table entry belonging to the current thread, and thread-cell-set! modifies it.

With thread cells, it is possible to implement dynamic binding correctly in the presence of multiprocessing: *dynamic-env*, instead of being bound directly to the environment, is now a thread cell:

```
(define *dynamic-env*
  (make-thread-cell (lambda (v) (cdr v))))

(define (make-fluid default) (cons 'fluid default))

(define (fluid-ref fluid)
  ((thread-cell-ref *dynamic-env*) fluid))

(define (bind-fluid fluid val thunk)
  (let ((old-env (thread-cell-ref *dynamic-env*))
        (new-env (shadow
                  (thread-cell-ref *dynamic-env*)
                  fluid val)))
    (dynamic-wind
     (lambda ()
       (thread-cell-set! *dynamic-env* new-env))
     thunk
     (lambda ()
       (thread-cell-set! *dynamic-env* old-env)))))
```

5.3 Modularity issues

While thread-local storage is a useful low-level abstraction, its use in programs imposes restrictions which may have an adverse effect on modularity. Consider the scenario from Section 3.3 with "dynamic binding/environment" replaced by "thread-local storage": module A creates and uses thread-local cells. This makes it much harder and potentially confusing for the client to use threads and control the sharing of among them. Here are the three choices for dynamic binding, revisited for thread-local storage:

1. New threads get a fresh thread-local store with default values for the thread-local variables—in this respect, they behave similarly to dynamic bindings.
2. Since thread-local storage is specifically not about binding, a `with-`*f*-like abstraction may not be feasible.
3. Inheritance of the thread-local storage is not easily possible for a new thread, as escape procedures do not capture the thread-local store.

Thus, if module A uses the thread-local store, the client has essentially no control over how A behaves with respect to the threads. This is unfortunate as the client might use threads for any number of reasons that in turn require different sharing semantics.

Especially the migration of escape procedures between threads raises troublesome questions with no obvious answer: As the escape procedure does not install the thread-local store from the thread which reified it, a solution to option #3—unsharing module A's state between the old and the new thread—becomes impossible. On the other hand, if the escape procedure were closed over the thread-local store, it would need to capture a *copy* of the store—otherwise, the name "thread-local storage" would be inappropriate, and the ensuing sharing semantics would carry more potential for confusion and error. Capturing the copy raises the next question: At what time should the program create the copy? At the time of capture, at the time of creating the new thread, or at the time of invocation of the escape procedure? The only feasible solution to the dilemma would be to make the thread-local store itself reifiable. However, it is unclear whether this abstraction would have benefits that outweigh the potential for confusion, and the inflexibility of abstractions which use thread-local storage in restricting ways.

Note that none of these problems manifest themselves in the implementation of dynamic binding presented in the previous section: the `dynamic-wind` thunks ensure that the `*dynamic-env*` thread-local-cell always holds the dynamic environment associated with the current continuation. Consequently, it seems that thread-local storage is a natural means for building other (still fairly low-level) abstractions such as dynamic binding, but rarely appropriate for use in higher-level abstractions or in applications.

6 Semantics

This section provides semantic specifications for a subset of Scheme with `dynamic-wind` and threads. We start with a version of the R^{5}RS denotational semantics which describes the behavior of `dynamic-wind`. We then formulate a transition semantics equivalent to the denotational semantics, which in turn forms the basis for a semantics for a concurrent version of the Scheme subset. This concurrent semantics specifies the interaction between `dynamic-wind` and threads. (We have also formulated a semantics which accounts for multiprocessing and for `thread-wind` which we have relegated to Appendix A. The appendix also contains an augmented version of the entire R^{5}RS semantics.) Moreover, we present a version of the denotational semantics with an explicit dynamic environment, and show that implementing the dynamic environment indirectly with dynamic assignment and `dynamic-wind` is indeed equivalent to propagating it directly in the semantics, thus demonstrating the utility of the semantics.

For the definition of our subset of Scheme, Mini-Scheme, we employ the same terminology, and, where possible, the same notation as R^{5}RS. (See Appendix D for details.) As compared to the language covered by the R^{5}RS semantics, a procedure has a fixed number of parameters and returns a single value, a procedure body consists of a single expression, procedures do not have an identifying location, evaluation is always left-to-right, and `if` forms always specify both branches. Mini-Scheme does, however, feature assignment, `call-with-current-continuation` and `dynamic-wind`. Here is the expression syntax of Mini-Scheme:

Exp $\longrightarrow$ K | I | (E$_0$ E*) | (lambda (I*) E$_0$)
| (if E$_0$ E$_1$ E$_2$) | (set! I E)

6.1 Denotational semantics

The semantic domains are analogous to those in R^{5}RS with changes according to the restrictions of Mini-Scheme—expression continuations always take one argument. The definition of $\mathcal{E}$* now needs special multi-argument *argument continuations*.

$\phi \in \mathtt{F}$	$= (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\kappa \in \mathtt{K}$	$= \mathtt{E} \to \mathtt{C}$	expression continuations
$\kappa' \in \mathtt{K'}$	$= \mathtt{E}^* \to \mathtt{C}$	argument continuations
$\omega \in \mathtt{P}$	$= (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{\mathit{root}\}$	dynamic points

In addition, P is the domain for *dynamic points* which are nodes in the control tree: *root* is the root node, and all other nodes consist of two thunks and a parent node. Figure 5 shows the semantics for Mini-Scheme expressions. It is completely analogous to the R^{5}RS version of the $\mathcal{E}$ function; the only addition is the propagation of the current dynamic point. The auxiliary functions are analogous to their R^{5}RS counterparts, apart from a change in *applicate* to take dynamic points into account:

$\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$\mathit{applicate} = \lambda\varepsilon\varepsilon^*\omega\kappa\,.\,\varepsilon \in \mathtt{F} \to (\varepsilon\,|\,\mathtt{F})\varepsilon^*\omega\kappa, \mathit{wrong}$ "bad procedure"

Here is a version of the *cwcc* primitive implementing `call-with-current-continuation` which respects `dynamic-wind`:

$\mathit{cwcc} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$ [call-with-current-continuation]
$\mathit{cwcc} =$
$\quad \mathit{onearg}\,(\lambda\varepsilon\omega\kappa\,.\,\varepsilon \in \mathtt{F} \to$
$\qquad\qquad (\mathit{applicate}\,\varepsilon$
$\qquad\qquad\qquad \langle(\lambda\varepsilon^*\omega'\kappa'\,.$
$\qquad\qquad\qquad\qquad \mathit{travel}\,\omega'\omega(\kappa(\varepsilon^* \downarrow 1)))\ \mathrm{in}\ \mathtt{E}\rangle$
$\qquad\qquad\qquad \omega\kappa),$
$\qquad\qquad \mathit{wrong}$ "bad procedure argument")

The escape procedure captures the dynamic point, and, when called, "travels" from the current dynamic point to it, running the *after* and *before* thunks in the process, before actually installing the continuation. Here is the definition of *travel*:

$\mathit{travel} : \mathtt{P} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}$
$\mathit{travel} = \lambda\omega_1\omega_2\,.\,\mathit{travelpath}\,(\mathit{path}\,\omega_1\omega_2)$

The *travelpath* function performs the actual travelling along a sequence of thunks and dynamic points, running each thunk with the

$$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{K}' \to \mathtt{C}$$

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\;(\lambda\varepsilon\,.\,\varepsilon = \mathit{undefined} \to \mathit{wrong}\text{ “undefined variable”},\; \mathit{send}\,\varepsilon\,\kappa)$$

$$\mathcal{E}[\![\texttt{(if E}_0\ \texttt{E}_1\ \texttt{E}_2\texttt{)}]\!] = \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\lambda\varepsilon\,.\,\mathit{truish}\,\varepsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,\; \mathcal{E}[\![\mathrm{E}_2]\!]\rho\omega\kappa)$$

$$\mathcal{E}[\![\texttt{(set! I E)}]\!] = \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,\omega\,(\lambda\varepsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\;\varepsilon\;(\mathit{send}\,\mathit{unspecified}\,\kappa))$$

$$\mathcal{E}[\![\texttt{(E}_0\ \texttt{E*)}]\!] = \lambda\rho\omega\kappa\,.\,\mathcal{E}^*(\langle \mathrm{E}_0\rangle \S\, \mathrm{E}^*)\rho\,\omega\,(\lambda\varepsilon^*\,.\,\mathit{applicate}\,(\varepsilon^* \downarrow 1)\,(\varepsilon^* \dagger 1)\,\omega\kappa)$$

$$\mathcal{E}[\![\texttt{(lambda (I*) E)}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{send}\,((\lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* = \#\mathrm{I}^* \to \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}[\![\mathrm{E}]\!]\rho'\omega'\kappa')\;(\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\;\varepsilon^*,\; \mathit{wrong}\text{ “wrong number of arguments”})\;\text{in E})\;\kappa$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\omega\kappa'\,.\,\kappa'\langle\,\rangle$$

$$\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}^*]\!] = \lambda\rho\omega\kappa'\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\lambda\varepsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\omega\,(\lambda\varepsilon^*\,.\,\kappa'\,(\langle\varepsilon_0\rangle \S\, \varepsilon^*)))$$

Figure 5. Semantics of Mini-Scheme expressions

corresponding dynamic point in place:

$$\begin{aligned}
&\mathit{travelpath} : (\mathtt{P}\times\mathtt{F})^* \to \mathtt{C} \to \mathtt{C}\\
&\mathit{travelpath} = \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,\\
&\qquad((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)\\
&\qquad\qquad(\lambda\varepsilon^*\,.\,\mathit{travelpath}\,(\pi^* \dagger 1)\theta)
\end{aligned}$$

The *path* function accepts two dynamic points and prefixes the journey between the two to its continuation argument:

$$\begin{aligned}
&\mathit{path} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{path} = \lambda\omega_1\omega_2\,.\,(\mathit{pathup}\,\omega_1(\mathit{commonancest}\,\omega_1\omega_2)) \S\\
&\qquad(\mathit{pathdown}\,(\mathit{commonancest}\,\omega_1\omega_2)\omega_2)
\end{aligned}$$

The *commonancest* function finds the lowest common ancestor of two dynamic points in the control tree. Leaving aside its definition for a moment, *pathup* ascends in the control tree, picking up *after* thunks, and *pathdown* descends, picking up *before* thunks:

$$\begin{aligned}
&\mathit{pathup} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{pathup} =\\
&\quad\lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad\langle(\omega_1, \omega_1 \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 2)\rangle \S\\
&\qquad(\mathit{pathup}\,(\omega_1 \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 3)\omega_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{pathdown} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{pathdown} =\\
&\quad\lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad(\mathit{pathdown}\,\omega_1(\omega_2 \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 3)) \S\\
&\qquad\langle(\omega_2, \omega_2 \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 1)\rangle
\end{aligned}$$

The *commonancest* function finds the lowest common ancestor of two dynamic points:

$$\begin{aligned}
&\mathit{commonancest} : \mathtt{P} \to \mathtt{P} \to \mathtt{P}\\
&\mathit{commonancest} =\\
&\quad\lambda\omega_1\omega_2\,.\,\text{the only element of}\\
&\qquad\{\omega' \mid \omega' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2),\\
&\qquad\quad\mathit{pointdepth}\,\omega' \geq \mathit{pointdepth}\,\omega''\\
&\qquad\quad\forall\omega'' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2)\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{pointdepth} : \mathtt{P} \to \mathtt{N}\\
&\mathit{pointdepth} =\\
&\quad\lambda\omega\,.\,\omega = \mathit{root} \to 0, 1 + (\mathit{pointdepth}\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 3))
\end{aligned}$$

The *ancestors* function computes the set of ancestors of a node (including the node itself):

$$\begin{aligned}
&\mathit{ancestors} : \mathtt{P} \to \mathcal{P}\mathtt{P}\\
&\mathit{ancestors} =\\
&\quad\lambda\omega\,.\,\omega = \mathit{root} \to \{\omega\}, \{\omega\} \cup (\mathit{ancestors}\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{P}) \downarrow 3))
\end{aligned}$$

The `dynamic-wind` primitive calls its first argument, then calls its second argument with a new node attached to the control tree, and then calls its third argument:

$$\begin{aligned}
&\mathit{dynamicwind} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&\mathit{dynamicwind} =\\
&\quad\mathit{threearg}\,(\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F} \wedge \varepsilon_3 \in \mathtt{F}) \to\\
&\qquad\mathit{applicate}\,\varepsilon_1\langle\rangle\omega(\lambda\zeta^*\,.\\
&\qquad\quad\mathit{applicate}\,\varepsilon_2\langle\rangle((\varepsilon_1 \mid \mathtt{F}, \varepsilon_3 \mid \mathtt{F}, \omega)\text{ in P})\\
&\qquad\qquad(\lambda\varepsilon^*\,.\,\mathit{applicate}\,\varepsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\varepsilon^*))),\\
&\qquad\mathit{wrong}\text{ “bad procedure argument”})
\end{aligned}$$

6.2 Transition Semantics

The denotational semantics is an awkward basis for incorporating concurrency. We therefore formulate a transition semantics [27] based on the CEK machine [6] based on the denotational semantics which is amenable to the addition of concurrency. Figure 6 shows the semantics. We deliberately use the functional environment and store *mutatis mutandis* and the same letters from the denotational semantics to simplify the presentation.

The $\longmapsto$ relation describes transitions between states. Two kinds of state exist: either the underlying machine is about to start evaluating an expression, or it must return a value to the current continuation. The former kind is represented by a tuple $\langle\sigma, \langle \mathrm{E}, \rho, \omega, \kappa\rangle\rangle$ where σ is the current store, E is the expression to be evaluated, ρ is the current environment, ω is the current dynamic point, and κ is the continuation of E. The latter kind of state is a tuple $\langle\sigma, \langle\kappa, \omega, \varepsilon\rangle\rangle$; σ, κ, and ω are as before, and ε is the value being passed to κ. The notable addition to the CEK machine is the `path` continuation which tracks the `dynamic-wind` *after* and *before* thunks that still need to run before returning to the “real” continuation.

6.3 Adding concurrency to the semantics

Figure 7 extends the sequential transition semantics by concurrency with preemptive scheduling in a way similar to the semantic specification of Concurrent ML [29]. The relation $\Longrightarrow$ operates on tuples, each of which consists of the global store and a *process set* containing the running threads. Each process is represented by a unique identifier ι and a state γ which is the state of the sequential semantics, sans the store. The *newid* function allocates an unused process identifier. The first rule adds concurrency. The second rule (added to the sequential semantics) describes the behavior of `spawn`: the program must first evaluate `spawn`'s argument and pass the result

$$
\begin{array}{rcll}
\sigma \in \mathtt{S}_d & = & \mathtt{L} \to (\mathtt{E}_d \times \mathtt{T}) & \omega \in \mathtt{P}_d = (\mathtt{F}_d \times \mathtt{F}_d \times \mathtt{P}_d) + \{\mathit{root}\} \\
\mathrm{State}_d & = & \mathtt{S}_d \times \mathrm{PState}_d & \phi \in \mathtt{F}_d = \langle \mathtt{cl}\ \rho, \mathrm{I}^*, \mathrm{E}\rangle \\
\mathrm{PState}_d & = & (\mathtt{E}_d \times \mathtt{U} \times \mathtt{P} \times \mathtt{K}_d) \mid (\mathtt{K}_d \times \mathtt{P}_d \times \mathtt{E}_d) & \varepsilon \in \mathtt{E}_d = \ldots \mid \mathtt{M} \mid \mathtt{F}_d \\
\kappa \in \mathtt{K}_d & = & \mathtt{stop} \mid \langle \mathtt{cnd}\ \mathrm{E}_1, \mathrm{E}_2, \rho, \kappa\rangle \mid \langle \mathtt{app}\ \langle \ldots, \varepsilon, \bullet, \mathrm{E}, \ldots\rangle, \rho, \omega, \kappa\rangle \mid \langle \mathtt{set!}\ \alpha, \kappa\rangle \mid \langle \mathtt{cwcc}\ \kappa\rangle \\
& \mid & \langle \mathtt{dw}\ \bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle \mid \langle \mathtt{dw}\ \varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa\rangle \mid \langle \mathtt{dw}\ \varepsilon_0, \varepsilon_1, \bullet, \rho, \omega, \kappa\rangle \\
& \mid & \langle \mathtt{dwe}\ \varepsilon_1, \varepsilon_2, \varepsilon_3, \rho, \omega, \kappa\rangle \mid \langle \mathtt{dwe}\ \varepsilon, \rho, \omega, \kappa\rangle \mid \langle \mathtt{return}\ \varepsilon, \kappa\rangle \mid \langle \mathtt{path}\ (\omega, \phi)^*, \varepsilon, \kappa\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \sigma, \langle \mathrm{I}, \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \sigma(\mathit{lookup}\,\rho\,\mathrm{I}) \downarrow 1\rangle\rangle \\
\langle \sigma, \langle \mathrm{K}, \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \mathcal{K}[\![\mathrm{K}]\!]\rangle\rangle \\
\langle \sigma, \langle (\mathtt{lambda}\ (\mathrm{I}^*)\ \mathrm{E}_0), \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \langle \mathtt{cl}\ \rho, \mathrm{I}^*, \mathrm{E}_0\rangle\rangle\rangle \\
\langle \sigma, \langle (\mathtt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2), \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{cnd}\ \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle (\mathrm{E}_0\ \mathrm{E}^*), \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{app}\ \langle \bullet, \mathrm{E}^*\rangle, \rho, \omega, \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle (\mathtt{set!}\ I\ E), \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle E, \rho, \omega, \langle \mathtt{set!}\ (\mathit{lookup}\,\rho\,I), \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle (\mathtt{call/cc}\ \mathrm{E}), \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle E, \rho, \omega, \langle \mathtt{cwcc}\ \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle (\mathtt{dynamic\text{-}wind}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2), \rho, \omega, \kappa\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{dw}\ \bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle\rangle\rangle \\
\\
\langle \sigma, \langle \langle \mathtt{cnd}\ \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle, \omega', \mathit{false}\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_2, \rho, \omega, \kappa\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{cnd}\ \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle, \omega', \varepsilon\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_1, \rho, \omega, \kappa\rangle\rangle \quad \text{if } \varepsilon \neq \mathit{false} \\
\langle \sigma, \langle \langle \mathtt{app}\ \langle \ldots, \varepsilon_i, \bullet, \mathrm{E}_{i+2}, \ldots\rangle, \rho, \omega, \kappa\rangle, \omega', \varepsilon_{i+1}\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_{i+2}, \rho, \omega, \langle \mathtt{app}\ \langle \ldots, \varepsilon_i, \varepsilon_{i+1}, \bullet, \ldots\rangle, \rho, \omega, \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{app}\ \langle \varepsilon_0, \ldots, \varepsilon_{n-1}, \bullet\rangle, \rho, \omega, \kappa\rangle, \omega', \varepsilon_n\rangle\rangle & \longmapsto & \langle \sigma[\varepsilon_1/\alpha_1]\ldots[\varepsilon_n/\alpha_n], \langle \mathrm{E}_0, \rho_0[\alpha_1/\mathrm{I}_1]\ldots[\alpha_n/\mathrm{I}_n], \omega, \kappa\rangle\rangle \\
& & \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}\ \rho_0, \langle \mathrm{I}_1, \ldots, \mathrm{I}_n\rangle, \mathrm{E}_0\rangle, \alpha_1 = \mathit{new}\,\sigma \mid \mathrm{L}, \alpha_2 = \mathit{new}\,\sigma[\varepsilon_1/\alpha_1] \mid \mathrm{L}, \ldots \\
\langle \sigma, \langle \langle \mathtt{app}\ \langle \varepsilon_0, \bullet\rangle, \rho, \omega, \kappa\rangle, \omega', \varepsilon_1\rangle\rangle & \longmapsto & \langle \sigma, \langle \langle \mathtt{path}\ (\mathit{path}\,\omega\omega'), \varepsilon_1, \kappa'\rangle, \omega, \mathit{unspecified}\rangle\rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cont}\ \omega', \kappa'\rangle \\
\langle \sigma, \langle \langle \mathtt{set!}\ \alpha, \kappa\rangle, \omega', \varepsilon\rangle\rangle & \longmapsto & \langle \sigma[\langle \varepsilon, \mathit{true}\rangle/\alpha], \langle \kappa, \omega, \mathit{unspecified}\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{cwcc}\ \kappa\rangle, \omega, \varepsilon\rangle\rangle & \longmapsto & \langle \sigma[\langle \mathtt{cont}\ \omega, \kappa\rangle/\mathit{new}\,\sigma], \langle \mathrm{E}_0, \rho_0[\mathit{new}\,\sigma/I], \omega, \kappa\rangle\rangle \quad \text{if } \varepsilon = \langle \mathtt{cl}\ \rho_0, \langle I\rangle, \mathrm{E}_0\rangle \\
\langle \sigma, \langle \langle \mathtt{dw}\ \bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa\rangle, \omega', \varepsilon_0\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_1, \rho, \omega, \langle \mathtt{dw}\ \varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{dw}\ \varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa\rangle, \omega', \varepsilon_1\rangle\rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_2, \rho, \omega, \langle \mathtt{dw}\ \varepsilon_1, \varepsilon_2, \bullet, \rho, \omega, \kappa\rangle\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{dw}\ \varepsilon_0, \varepsilon_1, \bullet, \rho, \omega, \kappa\rangle, \omega', \varepsilon_2\rangle\rangle & \longmapsto & \langle \sigma, \langle E_0, \rho_0, \omega, \langle \mathtt{dwe}\ \varepsilon_0, \varepsilon_1, \varepsilon_2, \rho, \omega, \kappa\rangle\rangle\rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}\ \rho_0, \langle\rangle, \mathrm{E}_0\rangle \\
\langle \sigma, \langle \langle \mathtt{dwe}\ \varepsilon_0, \varepsilon_1, \varepsilon_2, \rho, \omega, \kappa\rangle, \omega', \varepsilon_0'\rangle\rangle & \longmapsto & \langle \sigma, \langle E_1, \rho_1, (\varepsilon_0, \varepsilon_2, \omega), \langle \mathtt{dwe}\ \varepsilon_2, \rho, \omega, \kappa\rangle\rangle\rangle \quad \text{if } \varepsilon_1 = \langle \mathtt{cl}\ \rho_1, \langle\rangle, \mathrm{E}_1\rangle \\
\langle \sigma, \langle \langle \mathtt{dwe}\ \varepsilon_2, \rho, \omega, \kappa\rangle, \omega', \varepsilon_1'\rangle\rangle & \longmapsto & \langle \sigma, \langle E_2, \rho_2, \omega, \langle \mathtt{return}\ \varepsilon_1', \kappa\rangle\rangle\rangle \quad \text{if } \varepsilon_2 = \langle \mathtt{cl}\ \rho_2, \langle\rangle, \mathrm{E}_2\rangle \\
\langle \sigma, \langle \langle \mathtt{return}\ \varepsilon, \kappa\rangle, \omega', \varepsilon'\rangle\rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \varepsilon\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{path}\ \langle\rangle, \varepsilon, \kappa\rangle, \omega', \varepsilon'\rangle\rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \varepsilon\rangle\rangle \\
\langle \sigma, \langle \langle \mathtt{path}\ \langle(\omega_0, \varepsilon_0), (\omega_1, \varepsilon_1), \ldots\rangle, \varepsilon, \kappa\rangle, \omega', \varepsilon'\rangle\rangle & \longmapsto & \langle \sigma, \langle E_0, \rho_0, \omega_0, \langle \mathtt{path}\ \langle(\omega_1, \varepsilon_1), \ldots\rangle, \varepsilon, \kappa\rangle\rangle\rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}\ \rho_0, \langle\rangle, \mathrm{E}_0\rangle
\end{array}
$$

Figure 6. Transition semantics for Mini-Scheme

to the `spwn` continuation. Once that happens, the third rule describes the creation of a new thread with an empty control tree and an empty continuation. The last rule removes a thread from the system once it has reached the empty continuation.

6.4 Relating the semantics

To relate the operational and the denotational semantics, we first define an evaluation function for the transition semantics:

$$\mathit{eval}(E, \rho, \omega, \kappa, \sigma) = \varepsilon \quad \text{if } \langle \sigma, \langle E, \rho, \omega, \kappa\rangle\rangle \longmapsto^* \langle \sigma', \langle \mathtt{stop}, \omega', \varepsilon\rangle\rangle$$

To actually prove the evaluation functions equivalent, their arguments and the result need to be equivalent in some sense. We conjecture that defining relations between the semantic domains in the spirit of [30, Section 12.6] provides us with the right notion of equivalence. Using R_{cont} to relate continuations, R_{dp} for dynamic points, R_{store} for stores, and R_* for values including errors and $\bot$, the equation we would like to hold is:

PROPOSITION 1. *For any Mini-Scheme expression E and environment* ρ*, if* $\langle \hat{\kappa}, \kappa\rangle \in R_{cont}$*,* $\langle \hat{\omega}, \omega\rangle \in R_{dp}$*, and* $\langle \hat{\sigma}, \sigma\rangle \in R_{store}$*, then*

$$\langle \mathit{eval}(E, \rho, \hat{\omega}, \hat{\kappa}, \hat{\sigma}), \mathcal{E}[\![E]\!]\rho\omega\kappa\sigma\rangle \in R_*$$

We commit the actual definition of the relations and the proof of the proposition to future work.

6.5 Semantics for dynamic binding

This section extends the denotational semantics for Mini-Scheme with a dynamic environment. We use the denotational semantics for `dynamic-wind` to prove the indirect implementation of dynamic binding from Section 2.2 correct. The new semantics requires a dynamic environment domain and extends the semantic domain for procedures and dynamic points by a dynamic environment:

$\phi \in \mathtt{F} = (\mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\psi \in \mathtt{D} = \mathtt{E} \to \mathtt{E}$	dynamic environments
$\omega \in \mathtt{P} = (\mathtt{F} \times \mathtt{F} \times \mathtt{P} \times \mathtt{D}) + \{\mathit{root}\}$	dynamic points

The initial dynamic environment is $\psi_{\mathit{init}} = \lambda\varepsilon\,.\,(\varepsilon \mid \mathtt{E_p} \downarrow 2)$. The dynamic environment is threaded through the evaluation exactly like the dynamic point. (Revised evaluation functions are in Appendix B.) All previous definitions can be adapted *mutatis mutandis* except for *dynamicwind* which needs to insert the dynamic environment into this created point and *travelpath* which calls the thunks with the environment from the point:

$$
\begin{array}{l}
\mathit{dynamicwind} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C} \\
\mathit{dynamicwind} = \mathit{threearg} \\
\quad (\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\psi\kappa\,.\,(\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F} \wedge \varepsilon_3 \in \mathtt{F}) \to \\
\qquad \mathit{applicate}\,\varepsilon_1\langle\rangle\omega\psi \\
\qquad\qquad (\lambda\zeta^*\,.\,\mathit{applicate}\,\varepsilon_2\langle\rangle((\varepsilon_1 \mid \mathtt{F}, \varepsilon_3 \mid \mathtt{F}, \omega, \psi)\ \mathrm{in}\ \mathtt{P})\psi \\
\qquad\qquad\qquad (\lambda\varepsilon^*\,.\,\mathit{applicate}\,\varepsilon_3\langle\rangle\omega\psi(\lambda\zeta^*\,.\,\kappa\varepsilon^*))), \\
\qquad \mathit{wrong}\ \text{“bad procedure argument”}) \\
\\
\mathit{travelpath} : (\mathtt{P} \times \mathtt{F})^* \to \mathtt{C} \to \mathtt{C}
\end{array}
$$

$$\begin{array}{rcll} \iota & \in & \mathtt{I} & \text{process IDs} \\ \tau = \langle \iota, \gamma \rangle & \in & \mathtt{Z} = \mathtt{I} \times \mathrm{PState}_d & \text{processes} \\ \Psi & \in & \mathcal{P}^{\mathrm{fin}}\mathtt{I} & \text{process sets} \end{array}$$

$$\frac{\langle \sigma, \gamma \rangle \longmapsto \langle \sigma', \gamma' \rangle}{\langle \sigma, \Psi \cup \{\langle \iota, \gamma \rangle\} \rangle \Longrightarrow \langle \sigma', \Psi \cup \{\langle \iota, \gamma' \rangle\} \rangle}$$

$$\langle \sigma, \langle \texttt{(spawn E)}, \rho, \omega, \kappa \rangle \rangle \longmapsto \langle \sigma, \langle \mathrm{E}, \rho, \omega, \langle \texttt{spwn } \kappa \rangle \rangle \rangle$$

$$\langle \sigma, \Psi \rangle \Longrightarrow \langle \sigma, \Psi' \cup \{\langle \iota, \langle \kappa, \mathit{unspecified} \rangle \rangle, \langle \mathit{newid}\,\Psi, \langle E, \rho, \mathit{root}, \texttt{stop} \rangle \rangle\} \rangle \quad \text{if } \Psi = \Psi' \cup \{\langle \iota, \langle \texttt{spwn } \kappa, \varepsilon \rangle \rangle\},\ \varepsilon = \langle \texttt{cl } \rho, \langle \rangle, \mathrm{E} \rangle$$

$$\langle \sigma, \Psi \cup \{\langle \iota, \langle \texttt{stop}, \varepsilon \rangle \rangle\} \rangle \Longrightarrow \langle \sigma, \Psi \rangle$$

Figure 7. Concurrent evaluation

$\mathit{travelpath} = \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,$
$\quad ((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)(((\pi^* \downarrow 1) \downarrow 1) \downarrow 4)$
$\quad\quad (\lambda\varepsilon^*\,.\,\mathit{travelpath}\,(\pi^* \dagger 1)\theta)$

The only additions are the definitions for creating, referencing, and binding dynamic variables:

$\mathcal{E}_d[\![\texttt{(make-fluid E)}]\!] = \mathcal{E}_d[\![\texttt{(cons 'fluid E)}]\!]$

$\mathit{fluidref} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$
$\mathit{fluidref} = \mathit{onearg}\,(\lambda\varepsilon\omega\psi\kappa\,.\,\mathit{send}\,(\psi\,\varepsilon)\kappa)$

$\mathit{bindfluid} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$
$\mathit{bindfluid} =$
$\quad \mathit{threearg}\,(\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\psi\kappa\,.\,\varepsilon_3 \in \mathtt{F} \to (\varepsilon_3\,|\mathtt{F}|)\,\langle\rangle\,\sigma\,\psi[\varepsilon_1/\varepsilon_2]\,\kappa,$
$\quad\quad\quad

level continuations, and "cobs" ("continuation objects"), the actual escape procedures, which may perform work in addition to replacing the current meta-level continuation by another.

`Dynamic-wind` first appeared in the Scheme language definition in R^{5}RS. Sitaram, in the context of the *run* and *fcontrol* control operators, associates "prelude" and "postlude" procedures with each continuation delimiter. This mechanism is comparable to `dynamic-wind` [34]. Dybvig et al. also describe a similar but more general mechanism called *process filters* for subcontinuations [19]. Filinski uses `call/cc` to transparently implement layered monads [8]. He shows how to integrate multiple computational effects effects by defining the relevant operators in terms of `call/cc` and state, and then re-defining `call/cc` to be compatible with the new operators. Filinski notes that that this is similar in spirit to the redefinition of `call/cc` to accomodate `dynamic-wind`, and the goals of Filinski's work and of `dynamic-wind` are fundamentally similar.

The implementation of thread systems using `call/cc` goes back to Wand [37]. Haynes, Friedman, and others further develop this approach to implementing concurrency [17, 2]. It is also the basis for the implementation of Concurrent ML [29]. Shivers rectifies many of the misunderstandings concerning the relationship between (meta-level) continuations and threads [32]. Many implementors have since noted that `call/cc` is an appropriate explicative aid for understanding threads, but that it is not the right tool for implementing them, especially in the presence of `dynamic-wind`.

Dynamic binding goes back to early versions of Lisp [35]. Even though the replacement of dynamic binding by lexical binding was a prominent contribution of early Scheme, dynamic binding has found its way back into most implementations of Scheme and Lisp.

The inheritance issue for the dynamic environment also appears in the implementation of parallelism via futures, as noted in Feeley's Ph.D. thesis [3] and Moreau's work on the semantics of dynamic binding[26]. In the context of parallelism, inheritance is important because the `future` construct [15] is ideally a transparent annotation. This notion causes considerable complications for `call/cc`; Moreau investigates the semantical issues [25]. Inheritance is also a natural choice for concurrency in purely functional languages: in the Glasgow implementation of Concurrent Haskell, a new thread inherits the implicit parameters [24] from its parent. Most implementations of Common Lisp which support threads seem to have threads inherit the values of special (dynamically scoped) variables and share their values with all other threads.

The situation is different in concurrent implementations of Scheme: Scheme is not a purely functional language, and threads are typically not a transparent annotation for achieving parallelism. Therefore, Scheme implementations supporting threads and dynamic binding have made different choices: In MzScheme [9], fluid variables (called *parameters*) are inherited; mutations to parameters are only visible in the thread that performs them. The upcoming version of Gambit-C has inheritance, but parameters refer to shared cells [4]. Fluids in Scheme 48 [22] are not inherited, and do not support mutation. Scsh [33] supports a special kind of *thread fluid* [13] where inheritance can be specified upon creation. Discussion on the inheritance and sharing issues has often been controversial [4].

There is a considerable body of work on the interaction of parallelism and continuations (even though the term concurrency is often used): Parallel Scheme implementations have traditionally offered annotation-style abstractions for running computations on other processors, such as parallel procedure calls or futures [15]. These annotations are normally transparent in purely functional programs without `call/cc`. Implementors have tried to make them transparent even in the presence of `call/cc` [20], which makes it necessary (and sensible) to have reified continuations span multiple threads. However, none of the implementations behaves intuitively in all cases, and none maintains transparency when the program executes side effects. Hieb et al. [19] alleviate this problem by proposing the use of delimited continuations—so-called *subcontinuations*—to express intuitive behavior. All of this work is largely orthogonal to ours which is largely concerned with concurrency as a programming paradigm. However, in our view, this confirms our conclusion that comingling threads and continuations leads to undesirable complications.

8 Conclusion

Combining first-class continuations, `dynamic-wind`, dynamic binding, and concurrency in a single functional language is akin to walking a minefield. The design space exhibits many peculiarities, and its size is considerable; existing systems occupy different places within it. Some design choices lead to semantic or implementation difficulties, others impact the programmer's ability to write modular multithreaded programs. In general, the discussion about the correct way to combine these facilities has been plagued by controversy and confusion. In this paper, we have examined the interactions between them in a systematic way. The most important insights are:

- It is better to build first-class continuations and `dynamic-wind` on top of a native thread system rather than building the thread system on top of continuations.
- Decoupling threads from the sequential part of the programming language leads to clean semantic specifications and easier-to-understand program behavior.
- Abstractions for thread-aware programming are useful, but their use can have a negative impact on modularity and thus requires great care.
- The semantic interaction between threads and dynamic binding in Scheme is easiest to explain when newly created threads start with a fresh dynamic context. Even though this design option is not current practice in many systems, it also offers the greatest flexibility when writing modular abstractions which use threads and dynamic binding.

Our work opens a number of avenues for further research. In particular, an equational specification for `dynamic-wind` in the style of Felleisen and Hieb's framework [7] would be very useful. This could also be the basis for characterizing "benevolent" uses of `dynamic-wind` and `thread-wind` that do not interfere with `call/cc` in undesirable ways.

Acknowledgments: We thank Will Clinger, Olivier Danvy, Marc Feeley, Matthias Felleisen, and Matthew Flatt for patiently answering our questions. The reviewers of ICFP 2003 and the Scheme Workshop also provided helpful feedback.

9 References

[1] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, December 1998.

[2] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[3] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[4] Marc Feeley. SRFI 39: Parameter objects. `http://srfi.schemers.org/srfi-39/`, December 2002.

[5] Marc Feely. SRFI 18: Multithreading support. `http://srfi.schemers.org/srfi-18/`, March 2001.

[6] Matthias Felleisen and Daniel Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[8] Andrzej Filinski. Representing layered monads. In Alexander Aiken, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, USA, January 1999. ACM Press.

[9] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, January 2003. Version 203.

[10] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems. In Peter Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 138–147, Paris, France, September 1999. ACM Press, New York.

[11] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[12] Martin Gasbichler and Michael Sperber. Final shift for call/cc: Direct implementation of shift and reset. In Simon Peyton-Jones, editor, *Proc. International Conference on Functional Programming 2002*, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

[13] Martin Gasbichler and Michael Sperber. Processes vs. user-level threads in Scsh. In Olin Shivers, editor, *Proceedings of the Third Workshop on Scheme and Functional Programming*, Pittsburgh, October 2002.

[14] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming*, Lecture Notes in Computer Science, pages 122–136, Genova, Italy, April 2001. Springer-Verlag.

[15] Robert H. Halstead, Jr. A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[16] David R. Hanson and Todd A. Proebsting. Dynamic variables. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 264–273, Snowbird, UT, June 2001. ACM Press.

[17] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[18] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October 1987.

[19] Robert Hieb, R. Kent Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):x, 1994.

[20] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, Nice, France, 1990. ACM Press.

[21] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[22] Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at `http://www.s48.org/`.

[23] Richard Kelsey and Michael Sperber. SRFI 34: Exception handling for programs. `http://srfi.schemers.org/srfi-34/`, December 2002.

[24] Jeffery R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 108–118, Boston, MA, USA, January 2000. ACM Press.

[25] Luc Moreau. The semantics of future in the presence of first-class continuations and side-effects. Technical Report M95/3, University of Southampton, November 1995.

[26] Luc Moreau. A syntactic theory for dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[27] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[28] Christian Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In Philip Wadler, editor, *Proc. International Conference on Functional Programming 2000*, pages 23–33, Montreal, Canada, September 2000. ACM Press, New York.

[29] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[30] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[31] Olin Shivers. A Scheme Shell. Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1994.

[32] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes, Paris, France, January 1997. Dept. of Computer Science, Aarhus, Denmark.

[33] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. *Scsh Reference Manual*, 2003. Available from `http://www.scsh.net/`.

[34] Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, Rice University, Houston, Texas, April 1994.

[35] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In Jean E. Sammet, editor, *History of Programming*

Languages II, pages 231–270. ACM, New York, April 1993. SIGPLAN Notices 3(28).

[36] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[37] Mitchell Wand. Continuation-based multiprocessing. In J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, 1980. The Lisp Company.

A Multiprocessing and `Thread-wind`

Figure 8 shows how to extend the sequential transition semantics from Section 6.3 to account for multiprocessing and `thread-wind`: The $\longmapsto$ relation operates on machine states $\langle \sigma, \Psi, \Pi, \beta \rangle$. As before, σ is the global store, Ψ is still a process set, but contains only the threads currently running on a processor. Π is a *processor map* mapping a processor to a processor state, which is either `idle` or `running` ι for a processor running the thread with ID ι. β is the set of of idle threads waiting to be scheduled on a processor. Each member of this set consists of the thread ID, the dynamic point to return to, and the state of that thread.

The rules for running threads and spawning new ones are much as before, only extended to account for the new machine state components. (The *newid* function now takes both the active and idle process sets as arguments.) The last three rules control the swapping in and swapping out of threads: The first of these prepares a thread for swap-out, prefixing the current continuation with a winding path and a `suspend` marker. (For simplicity, we allow swapping out only when returning a value to a continuation.) The winding path is obtained by travelling up the control tree, only collecting *after* thunks introduced by `thread-wind`. (The new `P` domain distinguishes between nodes introduced by `dynamic-wind` and those introduced by `thread-wind` by a new boolean flag.) The subsequent rule actually performs the swapping out once the thread has reached the `suspend` marker. The last rule swaps a thread back in, prefixing the path back down to the target control node.

The `tpath` continuation works exactly the same as the `path` continuation, with the only exception that a processor running a thread in the midst of `tpath` continuation cannot swap that thread out.

B Semantics for Mini-Scheme with dynamic binding

Figure 9 describes evaluation functions $\mathcal{E}$ and $\mathcal{E}$* for Mini-Scheme with dynamic binding as described in Section 6.5.

C Defining `dynamic-wind` using the continuation monad

The published version of R^{5}RS says:

> The definition of `call-with-current-continuation` in Section 8.2 is incorrect because it is incompatible with `dynamic-wind`. As shown in Section 4 of [1], however, this incorrect semantics is adequate to define the `shift` and `reset` operators, which can then be used to define the correct semantics of both `dynamic-wind` and `call-with-current-continuation`.

The origin of this comment is unclear, and there is no published (or, to our knowledge, any) implementation of `call-with-current-continuation` and `dynamic-wind` to support this claim. We work out the details here. Our implementation represents a dynamic point as a pair of a pair of a *before* and an *after* thunk, and the parent point. The root point is represented as the empty list.

```
(define root-point '())

(define root-point? null?)

(define (make-point before after parent)
  (cons (cons before after) parent))

(define point-parent cdr)

(define (point-depth p)
  (if (root-point? p)
      0
      (+ 1 (point-depth (point-parent p)))))

(define point-before caar)

(define point-after cdar)
```

Filinski's framework for representing monads provides two functions `reify` and `reflect` which mediate between computations and values (the macro `reify*` simply wraps its argument into a thunk to shorten the rest of the examples):

```
(define (reflect meaning)
  (shift k (extend k meaning)))

(define (reify thunk)
  (reset (eta (thunk))))

(define-syntax reify*
  (syntax-rules ()
    ((reify* body ...)
     (reify (lambda () body ...)))))
```

See [12] for a Scheme version of Filinski's definition of `shift` and `reset` in terms of `call/cc`. The procedures `eta` and `extend` correspond to the usual monadic unit and extension functions. In Haskell, `eta` is known as `return` and `extend` as `bind` or the infix operator `>>=`.

Defining the continuation monad requires defining `eta` and `extend`. The datatype of the plain continuation monad contains a procedure which accepts a continuation as its argument and delivers its result by applying a continuation. The unit operation delivers a value by applying the continuation. The extension operation puts the function into the continuation:

```
(define (eta a)
  (lambda (c) (c a)))

(define (extend k m)
  (lambda (c)
    (m (lambda (v) ((k v) c)))))
```

For actually running programs, an evaluation function which supplies the identity function to its argument comes in handy:

```
(define (eval m)
  ((reify m) (lambda (v) v)))
```

$$
\begin{array}{rcll}
\omega & \in & \mathtt{P} = (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) + \{\mathit{root}\} & \text{dynamic points and thread points}\\
\varphi & \in & \mathtt{W} & \text{processor}\\
\varsigma & \in & \mathtt{J} = \mathtt{idle} \mid \mathtt{running\ I} & \text{processor state}\\
\Pi & \in & \mathtt{W} \to \mathtt{J} & \text{processor map}\\
\pi^* & \in & \mathtt{G} = (\mathtt{P}\times\mathtt{F})^* & \mathtt{thread\text{-}wind}\ \text{paths}\\
\beta & \in & \mathtt{B} = \mathcal{P}^{\mathrm{fin}}(\mathtt{I}\times\mathtt{P}\times\mathtt{Y}) & \text{idle threads}
\end{array}
$$

$$
\frac{\langle\sigma,\gamma\rangle \longmapsto \langle\sigma',\gamma'\rangle}{\langle\sigma,\Psi\cup\{\langle\iota,\gamma\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma',\Psi\cup\{\langle\iota,\gamma'\rangle\},\Pi,\beta\rangle}
$$

$$
\langle\sigma,\Psi\cup\{\langle\iota,\langle\mathtt{stop},\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi,\Pi[\mathtt{idle}/\varphi],\beta\rangle \quad \text{if } \Pi\varphi = \mathtt{running}\ \iota
$$

$$
\langle\sigma,\langle\mathtt{(spawn\ E)},\rho,\omega,\kappa\rangle,\Pi,\beta\rangle \Longmapsto \langle\sigma,\langle\mathrm{E},\rho,\omega,\langle\mathtt{spwn}\ \kappa\rangle\rangle,\Pi,\beta\rangle
$$

$$
\langle\sigma,\Psi,\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi'\cup\{\langle\iota,\langle\kappa,\mathit{unspecified}\rangle\rangle\},\beta\cup\{\langle\mathit{newid}\,\Psi\,\beta,\mathit{root},\langle E,\rho,\mathit{root},\mathtt{stop}\rangle\rangle\}\rangle
$$
$$
\text{if } \Psi = \Psi'\cup\{\langle\iota,\langle\mathtt{spwn}\ \kappa,\varepsilon\rangle\rangle\},\ \varepsilon = \langle\mathtt{cl}\ \rho,\langle\rangle,\mathrm{E}\rangle
$$

$$
\langle\sigma,\Psi\cup\{\langle\iota,\langle\kappa,\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{tpath}\ \ (\mathit{pathup}_m\,\omega),\varepsilon,\langle\mathtt{suspend}\ \kappa\rangle\rangle,\omega,\mathit{unspecified}\rangle\rangle\},\Pi,\beta\rangle
$$
$$
\text{if } \kappa \text{ does not contain } \mathtt{tpath}
$$

$$
\langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{suspend}\ \kappa\rangle,\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi,\Pi[\mathtt{idle}/\varphi],\beta\cup\{\langle\iota,\omega,\langle\kappa,\omega,\varepsilon\rangle\rangle\}\rangle \quad \text{if } \Pi\varphi = \mathtt{running}\ \iota
$$

$$
\langle\sigma,\Psi,\beta\cup\{\langle\iota,\omega,\langle\kappa,\omega,\varepsilon\rangle\rangle\}\rangle \Longmapsto \langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{tpath}\ \ (\mathit{pathdown}_m\,\omega),\varepsilon,\kappa\rangle,\omega,\mathit{unspecified}\rangle\rangle\},\Pi[\mathtt{running}\ \iota/\varphi],\beta\rangle \quad \text{if } \Pi\varphi = \mathtt{idle}
$$

$$
\begin{array}{rcll}
\langle\sigma,\langle\langle\mathtt{tpath}\ \ \langle\rangle,\varepsilon,\kappa\rangle,\omega',\varepsilon'\rangle\rangle & \longmapsto & \langle\sigma,\langle\kappa,\omega,\varepsilon\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tpath}\ \ \langle(\omega_0,\varepsilon_0),(\omega_1,\varepsilon_1),\ldots\rangle,\varepsilon,\kappa\rangle,\omega',\varepsilon'\rangle\rangle & \longmapsto & \langle\sigma,\langle E_0,\rho_0,\omega_0,\langle\mathtt{tpath}\ \ \langle(\omega_1,\varepsilon_1),\ldots\rangle,\varepsilon,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_0 = \langle\mathtt{cl}\ \rho_0,\langle\rangle,\mathrm{E}_0\rangle
\end{array}
$$

$$
\begin{array}{rcll}
\langle\sigma,\langle\mathtt{(thread\text{-}wind\ E_0\ E_1\ E_2)},\rho,\omega,\kappa\rangle\rangle & \longmapsto & \langle\sigma,\langle\mathrm{E}_0,\rho,\omega,\langle\mathtt{tw}\ \bullet,\mathrm{E}_1,\mathrm{E}_2,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \bullet,\mathrm{E}_1,\mathrm{E}_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0\rangle\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_1,\omega,\langle\mathtt{tw}\ \varepsilon_0,\bullet,\mathrm{E}_2,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \varepsilon_0,\bullet,\mathrm{E}_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_1\rangle\rangle & \longmapsto & \langle\sigma,\langle E_2,\rho_2,\omega,\langle\mathtt{tw}\ \varepsilon_1,\varepsilon_2,\bullet,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \varepsilon_0,\varepsilon_1,\bullet,\rho,\omega,\kappa\rangle,\omega',\varepsilon_2\rangle\rangle & \longmapsto & \langle\sigma,\langle E_0,\rho_0,\omega,\langle\mathtt{twe}\ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_0 = \langle\mathtt{cl}\ \rho_0,\langle\rangle,\mathrm{E}_0\rangle\\
\langle\sigma,\langle\langle\mathtt{dwe}\ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0'\rangle\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_0,(\varepsilon_0,\varepsilon_2,\mathit{false},\omega),\langle\mathtt{dwe}\ \varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_1 = \langle\mathtt{cl}\ \rho_1,\langle\rangle,\mathrm{E}_1\rangle\\
\langle\sigma,\langle\langle\mathtt{twe}\ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0'\rangle\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_0,(\varepsilon_0,\varepsilon_2,\mathit{true},\omega),\langle\mathtt{dwe}\ \varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_1 = \langle\mathtt{cl}\ \rho_1,\langle\rangle,\mathrm{E}_1\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{pathup}_m : \mathtt{P} \to \mathtt{G}\\
\mathit{pathup}_m =\\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to \langle\rangle,\\
\qquad (\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 3) = \mathit{true} \to \langle(\omega,\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 2)\rangle \S\, (\mathit{pathup}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4)),\\
\qquad\quad (\mathit{pathup}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4))\\
\mathit{pathdown}_m : \mathtt{P} \to \mathtt{G}\\
\mathit{pathdown}_m =\\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to \langle\rangle,\\
\qquad (\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 3) = \mathit{true} \to (\mathit{pathdown}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4)) \S\, \langle(\omega,\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 2)\rangle,\\
\qquad\quad (\mathit{pathdown}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4))
\end{array}
$$

Figure 8. Multiprocessor evaluation

The definition of `call/cc` is straightforward:

```
(define (call/cc h)
  (reflect
   (lambda (c)
     (let ((k (lambda (v)
                (reflect (lambda (c-prime) (c v))))))
       ((reify* (h k)) c)))))
```

To incorporate `dynamic-wind` we pair the continuation function with a dynamic point. `Eta` still applies the continuation to its argument, while `extend` supplies the same dynamic point to both of its arguments.

```
(define (eta a)
  (lambda (cdp) ((car cdp) a)))
```

```
(define (extend k m)
  (lambda (cdp)
    (m (cons (lambda (v) ((k v) cdp)) (cdr cdp)))))
```

The evaluation procedure takes a thunk representing the computation as argument, reifies it and applies it to the identity continuation and the root point:

```
(define (eval m)
  ((reify m) (cons (lambda (v) v) root-point)))
```

`Dynamic-wind` evaluates first evaluates the before thunk. It then evaluates the body thunk with a new dynamic point, before it evaluates the after thunk with a continuation which applies the continuation of the `dynamic-wind` to the result of the body.

$$\mathcal{E}_d : \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}_d{}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$$

$$\mathcal{E}_d[\![\mathrm{K}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\mathcal{E}_d[\![\mathrm{I}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\,(\lambda\varepsilon\,.\,\varepsilon = \mathit{undefined} \to \mathit{wrong}\text{ “undefined variable”},\ \mathit{send}\,\varepsilon\,\kappa)$$

$$\mathcal{E}_d[\![\texttt{(if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}_0]\!]\,\rho\omega\psi\,(\lambda\varepsilon\,.\,\mathit{truish}\,\varepsilon \to \mathcal{E}_d[\![\mathrm{E}_1]\!]\rho\omega\psi\kappa,\ \mathcal{E}_d[\![\mathrm{E}_2]\!]\rho\omega\psi\kappa)$$

$$\mathcal{E}_d[\![\texttt{(set! I E)}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}]\!]\,\rho\,\omega\psi\,(\lambda\varepsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\,\varepsilon\,(\mathit{send}\,\mathit{unspecified}\,\kappa))$$

$$\mathcal{E}_d[\![\texttt{(}\mathrm{E}_0\ \mathrm{E}^*\texttt{)}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d{}^*(\langle\mathrm{E}_0\rangle \S \mathrm{E}^*)\,\rho\,\omega\,\psi\,(\lambda\varepsilon^*\,.\,\mathit{applicate}\,(\varepsilon^* \downarrow 1)\,(\varepsilon^* \dagger 1)\,\omega\psi\kappa)$$

$$\mathcal{E}_d[\![\texttt{(lambda (I*) E)}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathit{send}\,((\lambda\varepsilon^*\omega'\psi'\kappa'\,.\,\#\varepsilon^* = \#\mathrm{I}^* \to \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}_d[\![\mathrm{E}]\!]\rho'\omega'\psi'\kappa')\,(\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\,\varepsilon^*,\ \mathit{wrong}\text{ “wrong number of arguments”})\ \text{in E})\,\kappa$$

$$\mathcal{E}_d{}^*[\![\,]\!] = \lambda\rho\omega\psi\kappa\,.\,\kappa\langle\rangle$$

$$\mathcal{E}_d{}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}_0]\!]\,\rho\omega\psi\,(\mathit{single}(\lambda\varepsilon_0\,.\,\mathcal{E}_d{}^*[\![\mathrm{E}^*]\!]\,\rho\omega\psi\,(\lambda\varepsilon^*\,.\,\kappa\,(\langle\varepsilon_0\rangle \S \varepsilon^*))))$$

Figure 9. Semantics of Mini-Scheme with dynamic environment

```
(define (dynamic-wind before thunk after)
  (reflect
   (lambda (cdp)
     ((reify* (before))
      (cons (lambda (v1)
              ((reify* (thunk))
               (cons (lambda (v2)
                       ((reify* (after))
                        (cons (lambda (v3)
                                ((car cdp) v2))
                              (cdr cdp))))
                     (make-point before after
                                 (cdr cdp)))))
            (cdr cdp))))))
```

`Call/cc` is responsible for generating an escape procedure which calls the appropriate set of before and after thunks. The following code defers this to the procedure `travel-to-point!`:

```
(define (call/cc h)
  (reflect
   (lambda (cdp)
     (let ((k (lambda (v)
                (reflect
                 (lambda (cdp-prime)
                   ((reify* (travel-to-point!
                             (cdr cdp-prime)
                             (cdr cdp)))
                    (cons (lambda (ignore)
                            ((car cdp) v))
                          (cdr cdp))))))))
       ((reify* (h k)) cdp)))))
```

`Travel-to-point!` implements an ingenious algorithm invented by Pavel Curtis for Scheme Xerox and used in Scheme 48:

```
(define (travel-to-point! here target)
  (cond ((eq? here target) 'done)
        ((or (root-point? here)
             (and (not (root-point? target))
                  (< (point-depth here)
                     (point-depth target))))
         (travel-to-point! here
                           (point-parent target))
         (with-point target
           (lambda () ((point-before target)))))
        (else
         (with-point here
           (lambda () ((point-after here))))
         (travel-to-point! (point-parent here)
                           target))))
```

The algorithm seeks the common ancestor by first walking up from lower of the two points until it is at the same level as the other. Then it alternately walks up one step at each of the points until it arrives at the same point, which is the common ancestor. The algorithms runs the *after* thunks walking up the source branch and winds up running the *before* thunks walking up the target branch. The helper procedure `with-point` takes a dynamic point and a thunk as its arguments and evaluates the thunk with the current continuation and the supplied point:

```
(define (with-point point thunk)
  (reflect
   (lambda (cdp)
     ((reify* (thunk))
      (cons (lambda (v) ((car cdp) v)) point)))))
```

D Denotational Semantics

[This is a version of the denotational semantics in R^{5}RS with `dynamic-wind`. We have copied the text verbatim, only making the necessary changes to account for the management of dynamic points.]

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [36]; the notation is summarized below:

$\langle\ldots\rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$t \to a, b$	McCarthy conditional “if t then a else b”
$\rho[x/i]$	substitution “ρ with x for i”
x in D	injection of x into domain D
$x \mid \mathtt{D}$	projection of x to domain D

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of pro-

cedure calls and multiple return values.

The boolean flag associated with pairs, vectors, and strings will be true for mutable objects and false for immutable objects.

The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute*, which must be inverses, to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

The storage allocator *new* is implementation-dependent, but it must obey the following axiom: if $new\,\sigma \in \mathtt{L}$, then $\sigma(new\,\sigma\,|\,\mathtt{L}) \downarrow 2 = false$.

The definition of $\mathcal{K}$ is omitted because an accurate definition of $\mathcal{K}$ would complicate the semantics without being very interesting.

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is

$$\mathcal{E}[\![\texttt{((lambda (I*) P')}\ \langle\text{undefined}\rangle\ \ldots\texttt{)}]\!]$$

where I* is the sequence of variables defined in P, P′ is the sequence of expressions obtained by replacing every definition in P by an assignment, ⟨undefined⟩ is an expression that evaluates to *undefined*, and $\mathcal{E}$ is the semantic function that assigns meaning to expressions.

D.1 Abstract syntax

$\mathtt{K} \in \mathtt{Con}$	constants, including quotations
$\mathtt{I} \in \mathtt{Ide}$	identifiers (variables)
$\mathtt{E} \in \mathtt{Exp}$	expressions
$\Gamma \in \mathtt{Com} = \mathtt{Exp}$	commands

$$\begin{aligned}
\mathtt{Exp} \longrightarrow\ & \mathtt{K} \mid \mathtt{I} \mid \mathtt{(}\mathtt{E}_0\ \mathtt{E*}\mathtt{)} \\
\mid\ & \mathtt{(lambda\ (I*)}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)} \\
\mid\ & \mathtt{(lambda\ (I*\ .\ I)}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)} \\
\mid\ & \mathtt{(lambda\ I}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)} \\
\mid\ & \mathtt{(if\ E}_0\ \mathtt{E}_1\ \mathtt{E}_2\mathtt{)} \mid \mathtt{(if\ E}_0\ \mathtt{E}_1\mathtt{)} \\
\mid\ & \mathtt{(set!\ I\ E)}
\end{aligned}$$

D.2 Domain equations

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\mathtt{T}$	$= \{false, true\}$	booleans
$\mathtt{Q}$		symbols
$\mathtt{H}$		characters
$\mathtt{R}$		numbers
$\mathtt{E}_p$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	pairs
$\mathtt{E}_v$	$= \mathtt{L*} \times \mathtt{T}$	vectors
$\mathtt{E}_s$	$= \mathtt{L*} \times \mathtt{T}$	strings
$\mathtt{M}$	$= \{false, true, null, undefined, unspecified\}$	miscellaneous
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E*} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\varepsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_p + \mathtt{E}_v + \mathtt{E}_s + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U} = \mathtt{Ide} \to \mathtt{L}$		environments
$\theta \in \mathtt{C} = \mathtt{S} \to \mathtt{A}$		command continuations
$\kappa \in \mathtt{K} = \mathtt{E*} \to \mathtt{C}$		expression continuations
$\mathtt{A}$		answers
$\mathtt{X}$		errors
$\omega \in \mathtt{P} = (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{root\}$		dynamic points

D.3 Semantic functions

$$\begin{aligned}
\mathcal{K} &: \mathrm{Con} \to \mathtt{E} \\
\mathcal{E} &: \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{E}^* &: \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{C} &: \mathrm{Com}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}
\end{aligned}$$

Definition of $\mathcal{K}$ deliberately omitted.

$$\mathcal{E}[\![\mathtt{K}]\!] = \lambda\rho\omega\kappa\,.\,send\,(\mathcal{K}[\![\mathtt{K}]\!])\,\kappa$$

$$\begin{aligned}
\mathcal{E}[\![\mathtt{I}]\!] = \lambda\rho\omega\kappa\,.\,&hold\,(lookup\,\rho\,\mathtt{I}) \\
&(single(\lambda\varepsilon\,.\,\varepsilon = undefined \to \\
&\qquad wrong\ \text{“undefined variable”}, \\
&\quad send\,\varepsilon\,\kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(E}_0\ \mathtt{E*)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}^*(permute(\langle \mathtt{E}_0 \rangle \S \mathtt{E*})) \\
&\qquad \rho \\
&\qquad \omega \\
&\qquad (\lambda\varepsilon^*\,.\,((\lambda\varepsilon^*\,.\,applicate\,(\varepsilon^* \downarrow 1)\,(\varepsilon^* \dagger 1)\,\omega\kappa) \\
&\qquad\quad (unpermute\,\varepsilon^*)))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(lambda\ (I*)}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathtt{L} \to \\
&\qquad\quad send\,(\langle new\,\sigma\,|\,\mathtt{L}, \\
&\qquad\qquad \lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* = \#\mathtt{I*} \to \\
&\qquad\qquad\qquad tievals(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathtt{E}_0]\!]\rho'\omega'\kappa')) \\
&\qquad\qquad\qquad\qquad (extends\,\rho\,\mathtt{I*}\,\alpha^*)) \\
&\qquad\qquad\qquad\quad \varepsilon^*, \\
&\qquad\qquad\qquad wrong\ \text{“wrong number of arguments”}\rangle \\
&\qquad\qquad \text{in } \mathtt{E}) \\
&\qquad\quad \kappa \\
&\qquad\quad (update\,(new\,\sigma\,|\,\mathtt{L})\,unspecified\,\sigma), \\
&\qquad wrong\ \text{“out of memory”}\,\sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(lambda\ (I*\ .\ I)}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathtt{L} \to \\
&\qquad\quad send\,(\langle new\,\sigma\,|\,\mathtt{L}, \\
&\qquad\qquad \lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* \geq \#\mathtt{I*} \to \\
&\qquad\qquad\qquad tievalsrest \\
&\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathtt{E}_0]\!]\rho'\omega'\kappa')) \\
&\qquad\qquad\qquad\qquad\quad (extends\,\rho\,(\mathtt{I*} \S \langle \mathtt{I} \rangle)\,\alpha^*)) \\
&\qquad\qquad\qquad\quad \varepsilon^* \\
&\qquad\qquad\qquad\quad (\#\mathtt{I*}), \\
&\qquad\qquad\qquad wrong\ \text{“too few arguments”}\rangle \text{ in } \mathtt{E}) \\
&\qquad\quad \kappa \\
&\qquad\quad (update\,(new\,\sigma\,|\,\mathtt{L})\,unspecified\,\sigma), \\
&\qquad wrong\ \text{“out of memory”}\,\sigma
\end{aligned}$$

$$\mathcal{E}[\![\mathtt{(lambda\ I}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)}]\!] = \mathcal{E}[\![\mathtt{(lambda\ (.\ I)}\ \Gamma\mathtt{*}\ \mathtt{E}_0\mathtt{)}]\!]$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(if\ E}_0\ \mathtt{E}_1\ \mathtt{E}_2\mathtt{)}]\!] = \\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathtt{E}_0]\!]\,\rho\omega\,(single\,(\lambda\varepsilon\,.\,truish\,\varepsilon \to \mathcal{E}[\![\mathtt{E}_1]\!]\rho\omega\kappa, \\
&\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathtt{E}_2]\!]\rho\omega\kappa))
\end{aligned}$$

$$\mathcal{E}[\![\texttt{(if E}_0\ \texttt{E}_1\texttt{)}]\!] =$$
$$\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\texttt{E}_0]\!]\,\rho\omega\,(\mathit{single}\,(\lambda\varepsilon\,.\,\mathit{truish}\,\varepsilon \to \mathcal{E}[\![\texttt{E}_1]\!]\rho\omega\kappa,$$
$$\qquad\qquad \mathit{send}\,\mathit{unspecified}\,\kappa))$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified*.

$$\mathcal{E}[\![\texttt{(set! I E)}]\!] =$$
$$\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\texttt{E}]\!]\,\rho\,\omega\,(\mathit{single}(\lambda\varepsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\texttt{I})$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \varepsilon$$
$$\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa)))$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\omega\kappa\,.\,\kappa\langle\,\rangle$$

$$\mathcal{E}^*[\![\texttt{E}_0\ \texttt{E}^*]\!] =$$
$$\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\texttt{E}_0]\!]\,\rho\omega\,(\mathit{single}(\lambda\varepsilon_0\,.\,\mathcal{E}^*[\![\texttt{E}^*]\!]\,\rho\omega\,(\lambda\varepsilon^*\,.\,\kappa\,(\langle\varepsilon_0\rangle \S\,\varepsilon^*))))$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\omega\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\omega\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\omega\,(\lambda\varepsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\omega\theta)$$

D.4 Auxiliary functions

$$\mathit{lookup} : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$$
$$\mathit{lookup} = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}$$

$$\mathit{extends} : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$$
$$\mathit{extends} =$$
$$\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho,$$
$$\qquad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$$

$$\mathit{wrong} : \mathtt{X} \to \mathtt{C} \qquad \text{[implementation-dependent]}$$

$$\mathit{send} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{send} = \lambda\varepsilon\kappa\,.\,\kappa\langle\varepsilon\rangle$$

$$\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$$
$$\mathit{single} =$$
$$\quad \lambda\psi\varepsilon^*\,.\,\#\varepsilon^* = 1 \to \psi(\varepsilon^* \downarrow 1),$$
$$\qquad \mathit{wrong}\ \text{“wrong number of return values”}$$

$$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\}) \qquad \text{[implementation-dependent]}$$

$$\mathit{hold} : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{hold} = \lambda\alpha\kappa\sigma\,.\,\mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma$$

$$\mathit{assign} : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$$
$$\mathit{assign} = \lambda\alpha\varepsilon\theta\sigma\,.\,\theta(\mathit{update}\,\alpha\varepsilon\sigma)$$

$$\mathit{update} : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$$
$$\mathit{update} = \lambda\alpha\varepsilon\sigma\,.\,\sigma[\langle\varepsilon, \mathit{true}\rangle/\alpha]$$

$$\mathit{tievals} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$$
$$\mathit{tievals} =$$
$$\quad \lambda\psi\varepsilon^*\sigma\,.\,\#\varepsilon^* = 0 \to \psi\langle\,\rangle\sigma,$$
$$\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \mathit{tievals}\,(\lambda\alpha^*\,.\,\psi(\langle \mathit{new}\,\sigma \mid \mathtt{L}\rangle \S\,\alpha^*))$$
$$\qquad\qquad\qquad\qquad (\varepsilon^* \dagger 1)$$
$$\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\varepsilon^* \downarrow 1)\sigma),$$
$$\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma$$

$$\mathit{tievalsrest} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}$$
$$\mathit{tievalsrest} =$$
$$\quad \lambda\psi\varepsilon^*\nu\,.\,\mathit{list}\,(\mathit{dropfirst}\,\varepsilon^*\nu)$$
$$\qquad\qquad (\mathit{single}(\lambda\varepsilon\,.\,\mathit{tievals}\,\psi\,((\mathit{takefirst}\,\varepsilon^*\nu) \S\,\langle\varepsilon\rangle)))$$

$$\mathit{dropfirst} = \lambda l n\,.\,n = 0 \to l, \mathit{dropfirst}\,(l \dagger 1)(n-1)$$

$$\mathit{takefirst} = \lambda l n\,.\,n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \S\,(\mathit{takefirst}\,(l \dagger 1)(n-1))$$

$$\mathit{truish} : \mathtt{E} \to \mathtt{T}$$
$$\mathit{truish} = \lambda\varepsilon\,.\,\varepsilon = \mathit{false} \to \mathit{false}, \mathit{true}$$

$$\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^* \qquad \text{[implementation-dependent]}$$

$$\mathit{unpermute} : \mathtt{E}^* \to \mathtt{E}^* \qquad \text{[inverse of } \mathit{permute}\text{]}$$

$$\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{applicate} =$$
$$\quad \lambda\varepsilon\varepsilon^*\omega\kappa\,.\,\varepsilon \in \mathtt{F} \to (\varepsilon \mid \mathtt{F} \downarrow 2)\varepsilon^*\omega\kappa, \mathit{wrong}\ \text{“bad procedure”}$$

$$\mathit{onearg} : (\mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$\mathit{onearg} =$$
$$\quad \lambda\zeta\varepsilon^*\omega\kappa\,.\,\#\varepsilon^* = 1 \to \zeta(\varepsilon^* \downarrow 1)\omega\kappa,$$
$$\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}$$

$$\mathit{twoarg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$\mathit{twoarg} =$$
$$\quad \lambda\zeta\varepsilon^*\omega\kappa\,.\,\#\varepsilon^* = 2 \to \zeta(\varepsilon^* \downarrow 1)(\varepsilon^* \downarrow 2)\omega\kappa,$$
$$\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}$$

$$\mathit{threearg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$\mathit{threearg} =$$
$$\quad \lambda\zeta\varepsilon^*\omega\kappa\,.\,\#\varepsilon^* = 3 \to \zeta(\varepsilon^* \downarrow 1)(\varepsilon^* \downarrow 2)(\varepsilon^* \downarrow 3)\omega\kappa,$$
$$\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}$$

$$\mathit{list} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{list} =$$
$$\quad \lambda\varepsilon^*\omega\kappa\,.\,\#\varepsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa,$$
$$\qquad\quad \mathit{list}\,(\varepsilon^* \dagger 1)(\mathit{single}(\lambda\varepsilon\,.\,\mathit{cons}\langle\varepsilon^* \downarrow 1, \varepsilon\rangle\kappa))$$

$$\mathit{cons} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{cons} =$$
$$\quad \mathit{twoarg}\,(\lambda\varepsilon_1\varepsilon_2\kappa\omega\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to$$
$$\qquad\qquad (\lambda\sigma'\,.\,\mathit{new}\,\sigma' \in \mathtt{L} \to$$
$$\qquad\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \mathit{new}\,\sigma' \mid \mathtt{L}, \mathit{true}\rangle$$
$$\qquad\qquad\qquad \text{in E})$$
$$\qquad\qquad\qquad \kappa$$
$$\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\varepsilon_2\sigma'),$$
$$\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma')$$
$$\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\varepsilon_1\sigma),$$
$$\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma)$$

$$\mathit{less} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{less} =$$
$$\quad \mathit{twoarg}\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{R} \wedge \varepsilon_2 \in \mathtt{R}) \to$$
$$\qquad\qquad \mathit{send}\,(\varepsilon_1 \mid \mathtt{R} < \varepsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa,$$
$$\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to <”})$$

$$\mathit{add} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{add} =$$
$$\quad \mathit{twoarg}\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{R} \wedge \varepsilon_2 \in \mathtt{R}) \to$$
$$\qquad\qquad \mathit{send}\,((\varepsilon_1 \mid \mathtt{R} + \varepsilon_2 \mid \mathtt{R})\ \text{in E})\kappa,$$
$$\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to +”})$$

$$\mathit{car} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{car} =$$
$$\quad \mathit{onearg}\,(\lambda\varepsilon\omega\kappa\,.\,\varepsilon \in \mathtt{E}_\mathrm{p} \to \mathit{car\text{-}internal}\,\varepsilon\kappa,$$
$$\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to }\texttt{car}\text{”})$$

$$\mathit{car\text{-}internal} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{car\text{-}internal} = \quad \lambda\varepsilon\omega\kappa\,.\,\mathit{hold}\,(\varepsilon \mid \mathtt{E}_\mathrm{p} \downarrow 1)\kappa$$

$cdr : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$ [similar to *car*]

$cdr\text{-}internal : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$ [similar to *car-internal*]

$$
\begin{array}{l}
setcar : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
setcar = \\
\quad twoarg\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,\varepsilon_1 \in \mathtt{E_p} \to \\
\qquad\qquad (\varepsilon_1 \mid \mathtt{E_p} \downarrow 3) \to assign\,(\varepsilon_1 \mid \mathtt{E_p} \downarrow 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \varepsilon_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad (send\,unspecified\,\kappa), \\
\qquad\qquad wrong\text{ “immutable argument to }\texttt{set-car!}\text{”}, \\
\qquad\qquad wrong\text{ “non-pair argument to }\texttt{set-car!}\text{”})
\end{array}
$$

$$
\begin{array}{l}
eqv : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
eqv = \\
\quad twoarg\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{M} \wedge \varepsilon_2 \in \mathtt{M}) \to \\
\qquad\qquad send\,(\varepsilon_1 \mid \mathtt{M} = \varepsilon_2 \mid \mathtt{M} \to true, false)\kappa, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{Q} \wedge \varepsilon_2 \in \mathtt{Q}) \to \\
\qquad\qquad\quad send\,(\varepsilon_1 \mid \mathtt{Q} = \varepsilon_2 \mid \mathtt{Q} \to true, false)\kappa, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{H} \wedge \varepsilon_2 \in \mathtt{H}) \to \\
\qquad\qquad\quad send\,(\varepsilon_1 \mid \mathtt{H} = \varepsilon_2 \mid \mathtt{H} \to true, false)\kappa, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{R} \wedge \varepsilon_2 \in \mathtt{R}) \to \\
\qquad\qquad\quad send\,(\varepsilon_1 \mid \mathtt{R} = \varepsilon_2 \mid \mathtt{R} \to true, false)\kappa, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{E_p} \wedge \varepsilon_2 \in \mathtt{E_p}) \to \\
\qquad\qquad\quad send\,((\lambda p_1 p_2\,.\,((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge \\
\qquad\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to true, \\
\qquad\qquad\qquad\qquad\quad false) \\
\qquad\qquad\qquad\qquad (\varepsilon_1 \mid \mathtt{E_p}) \\
\qquad\qquad\qquad\qquad (\varepsilon_2 \mid \mathtt{E_p})) \\
\qquad\qquad\qquad\qquad \kappa, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{E_v} \wedge \varepsilon_2 \in \mathtt{E_v}) \to \ldots, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{E_s} \wedge \varepsilon_2 \in \mathtt{E_s}) \to \ldots, \\
\qquad\qquad (\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F}) \to \\
\qquad\qquad\quad send\,((\varepsilon_1 \mid \mathtt{F} \downarrow 1) = (\varepsilon_2 \mid \mathtt{F} \downarrow 1) \to true, false) \\
\qquad\qquad\qquad\quad \kappa, \\
\qquad\qquad\quad send\,false\,\kappa)
\end{array}
$$

$$
\begin{array}{l}
apply : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
apply = \\
\quad twoarg\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,\varepsilon_1 \in \mathtt{F} \to valueslist\,\varepsilon_2(\lambda\varepsilon^*\,.\,applicate\,\varepsilon_1\varepsilon^*\omega\kappa), \\
\qquad\qquad wrong\text{ “bad procedure argument to }\texttt{apply}\text{”})
\end{array}
$$

$$
\begin{array}{l}
valueslist : \mathtt{E} \to \mathtt{K} \to \mathtt{C} \\
valueslist = \\
\quad \lambda\varepsilon\kappa\,.\,\varepsilon \in \mathtt{E_p} \to \\
\qquad\quad cdr\text{-}internal\,\varepsilon \\
\qquad\qquad\qquad (\lambda\varepsilon^*\,.\,valueslist \\
\qquad\qquad\qquad\qquad \varepsilon^* \\
\qquad\qquad\qquad\qquad (\lambda\varepsilon^*\,.\,car\text{-}internal \\
\qquad\qquad\qquad\qquad\qquad\quad \varepsilon \\
\qquad\qquad\qquad\qquad\qquad\quad (single(\lambda\varepsilon\,.\,\kappa(\langle\varepsilon\rangle \S \varepsilon^*))))), \\
\qquad \varepsilon = null \to \kappa\langle\rangle, \\
\qquad\quad wrong\text{ “non-list argument to }\texttt{values-list}\text{”}
\end{array}
$$

$$
\begin{array}{l}
cwcc : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \quad \texttt{[call-with-current-continuation]} \\
cwcc = \\
\quad onearg\,(\lambda\varepsilon\omega\kappa\,.\,\varepsilon \in \mathtt{F} \to \\
\qquad\qquad (\lambda\sigma\,.\,new\,\sigma \in \mathtt{L} \to \\
\qquad\qquad\quad applicate\,\varepsilon \\
\qquad\qquad\qquad\qquad \langle\langle new\,\sigma \mid \mathtt{L}, \\
\qquad\qquad\qquad\qquad\quad \lambda\varepsilon^*\omega'\kappa'\,.\,travel\,\omega'\omega(\kappa\varepsilon^*)\rangle \\
\qquad\qquad\qquad\qquad\quad \text{in } \mathtt{E}\rangle \\
\qquad\qquad\qquad\qquad \omega \\
\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad (update\,(new\,\sigma \mid \mathtt{L})
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad unspecified \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sigma), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad wrong\text{ “out of memory”}\,\sigma), \\
\qquad\qquad\qquad\qquad\qquad\qquad wrong\text{ “bad procedure argument”})
\end{array}
$$

$$
\begin{array}{l}
travel : \mathtt{P} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C} \\
travel = \\
\quad \lambda\omega_1\omega_2\,.\,travelpath\,((pathup\,\omega_1(commonancest\,\omega_1\omega_2)) \S \\
\qquad\qquad\qquad\qquad (pathdown\,(commonancest\,\omega_1\omega_2)\omega_2))
\end{array}
$$

$$
\begin{array}{l}
pointdepth : \mathtt{P} \to \mathtt{N} \\
pointdepth = \\
\quad \lambda\omega\,.\,\omega = root \to 0, 1 + (pointdepth\,(\omega \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{array}
$$

$$
\begin{array}{l}
ancestors : \mathtt{P} \to \mathcal{P}\mathtt{P} \\
ancestors = \\
\quad \lambda\omega\,.\,\omega = root \to \{\omega\}, \{\omega\} \cup (ancestors\,(\omega \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{array}
$$

$$
\begin{array}{l}
commonancest : \mathtt{P} \to \mathtt{P} \to \mathtt{P} \\
commonancest = \\
\quad \lambda\omega_1\omega_2\,.\,\text{the only element of} \\
\qquad\quad \{\omega' \mid \omega' \in (ancestors\,\omega_1) \cap (ancestors\,\omega_2), \\
\qquad\qquad pointdepth\,\omega' \geq pointdepth\,\omega'' \\
\qquad\qquad \forall\omega'' \in (ancestors\,\omega_1) \cap (ancestors\,\omega_2)\}
\end{array}
$$

$$
\begin{array}{l}
pathup : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^* \\
pathup = \\
\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle, \\
\qquad\quad \langle(\omega_1, \omega_1 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 2)\rangle \S \\
\qquad\quad (pathup\,(\omega_1 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3)\omega_2)
\end{array}
$$

$$
\begin{array}{l}
pathdown : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^* \\
pathdown = \\
\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle, \\
\qquad\quad (pathdown\,\omega_1(\omega_2 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3)) \S \\
\qquad\quad \langle(\omega_2, \omega_2 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 1)\rangle
\end{array}
$$

$$
\begin{array}{l}
travelpath : (\mathtt{P} \times \mathtt{F})^* \to \mathtt{C} \to \mathtt{C} \\
travelpath = \\
\quad \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta, \\
\qquad ((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1) \\
\qquad\qquad\qquad (\lambda\varepsilon^*\,.\,travelpath\,(\pi^* \dagger 1)\theta)
\end{array}
$$

$$
\begin{array}{l}
dynamicwind : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
dynamicwind = \\
\quad threearg\,(\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F} \wedge \varepsilon_3 \in \mathtt{F}) \to \\
\qquad\quad applicate\,\varepsilon_1\langle\rangle\omega(\lambda\zeta^*\,. \\
\qquad\qquad\quad applicate\,\varepsilon_2\langle\rangle((\varepsilon_1 \mid \mathtt{F}, \varepsilon_3 \mid \mathtt{F}, \omega)\text{ in }\mathtt{P}) \\
\qquad\qquad\qquad\quad (\lambda\varepsilon^*\,.\,applicate\,\varepsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\varepsilon^*))), \\
\qquad\quad wrong\text{ “bad procedure argument”})
\end{array}
$$

$$
\begin{array}{l}
values : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
values = \lambda\varepsilon^*\omega\kappa\,.\,\kappa\varepsilon^*
\end{array}
$$

$$
\begin{array}{l}
cwv : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \quad \texttt{[call-with-values]} \\
cwv = \\
\quad twoarg\,(\lambda\varepsilon_1\varepsilon_2\omega\kappa\,.\,applicate\,\varepsilon_1\langle\rangle\omega(\lambda\varepsilon^*\,.\,applicate\,\varepsilon_2\,\varepsilon^*\omega))
\end{array}
$$

Unwind-protect in portable Scheme

Dorai Sitaram
Verizon
40 Sylvan Road
Waltham, MA 02451

Abstract

Programming languages that allow non-local control jumps also need to provide an *unwind-protect* facility. Unwind-protect associates a *postlude* with a given block *B* of code, and guarantees that the postlude will always be executed regardless of whether *B* concludes normally or is exited by a jump. This facility is routinely provided in all languages with *first*-order control operators. Unfortunately, in languages such as Scheme and ML with *higher*-order control, unwind-protect does not have a clear meaning, although the need for some form of protection continues to exist. We will explore the problem of specifying and implementing unwind-protect in the higher-order control scenario of Scheme.

1 Introduction

Unwind-protect has a straightforward semantics for programming languages where non-local control jumps are purely first-order, i.e., computations can abort to a dynamically enclosing context, but can never re-enter an already exited context. Such languages include Common Lisp, Java, C++, and even text-editor languages like Emacs and Vim; all of them provide unwind-protect. An unwind-protected block of code *B* has a postlude *P* that is guaranteed to run whenever and however *B* exits, whether normally or via a non-local exit to some enclosing dynamic context. This is a useful guarantee to have, as we can have *P* encode *clean-up* actions that we can rely upon to happen. The canonical use for unwind-protect is to ensure that file ports opened in *B* get closed when *B* is exited.

```
(let ([o #f])
  (unwind-protect
    ;protected code
    (begin
      (set! o (open-output-file "file"))
      ... <possible non-local exit> ...
      )
    ;the postlude
    (close-output-port o)))
```

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Dorai Sitaram.

When we move to higher-order control scenarios such as Scheme [8] and ML [1], it is no longer clear what unwind-protect should mean. Here, control *can* re-enter a previously exited dynamic context, opening up new questions such as:

1. Should a prelude be considered in addition to the postlude?
2. Can the postlude be evaluated more than once?
3. Should the postlude be enabled only for some exits but not for others, and if so which?

The language Scheme provides a related operator called *dynamic-wind* that attaches a prelude and a postlude to a code block, and ensures that the postlude (prelude) is always evaluated whenever control exits (enters) the block. While this may seem like a natural extension of the first-order unwind-protect to a higher-order control scenario, it does not tackle the pragmatic need that unwind-protect addresses, namely, the need to ensure that a kind of "clean-up" happens only for those jumps that *significantly exit* the block, and not for those that are *minor excursions*. The crux is identifying which of these two categories a jump falls into, and perhaps allowing the user a way to explicitly fix the category. It usually makes no sense to re-enter a block after the clean-up has been performed (as in the port-closing example above): Thus there is no need for a specific prelude syntax beyond sequencing, and postludes need happen only once. Thus we can answer questions 1 and 2 above with No, but there is no single objectively correct answer to question 3.

2 Call/cc and how to constrain it

Scheme's control operator, *call-with-current-continuation* (abbreviated *call/cc*), allows control to transfer to arbitrary points in the program, not just to dynamically enclosing contexts. It does so by providing the user with a *continuation*, i.e., a procedural representation of the current control context, or more simply, "the rest of the program". Invoking this continuation at any point in the program causes that point's current context to be replaced by the context that the continuation represents. The user sees *call/cc* as a procedure that takes a single unary procedure *f* as argument. *f* is called with the current continuation (hence the operator's name). This continuation is a procedure that takes a single argument, which it inserts into the old program context.[1]

This ability to substitute the current program context by a previously captured snapshot of a program context is simple and powerful [6, 7, 10], but too low-level to be used straightaway for user-level

[1] I will ignore the presence of Scheme's multiple values, as they add no particular illumination to the problem we are addressing.

abstractions. In addition to the difficulty of encoding user abstractions in terms of *call/cc*, one must also ensure that the abstractions so defined can function without interference from other uses of *call/cc*. To solve this problem, Friedman and Haynes [5] illustrate a technique for *constraining* raw *call/cc*. They define new *call/cc* operators that call the original *call/cc*, but instead of directly calling the *call/cc*-argument on the continuation, they call it on a *continuation object* or *cob*, which is a procedure that performs whatever additional constraining tasks are required before calling the actual continuation.

Let us illustrate the cob technique to solve an easily described problem, that of fluid variables. The form **let-fluid** temporarily extends the fluid environment, **fluid-env**, for the duration of its dynamic extent.

```
(define *fluid-env* '())

(define-syntax let-fluid
  (syntax-rules ()
    [(let-fluid ((x e) ...) b ...)
     (let ([old-fluid-env *fluid-env*])
       (set! *fluid-env*
         (append! (list (cons 'x e) ...) *fluid-env*))
       (let ([result (begin b ...)])
         (set! *fluid-env* old-fluid-env)
         result))])))
```

Fluid variables are accessed using the form **fluid**, defined globally using **define-fluid**, and side-effected using **set-fluid!**:

```
(define-syntax fluid
  (syntax-rules ()
    [(fluid x)
     (cond [(assq 'x *fluid-env*) => cdr]
           [else (error 'undefined-fluid 'x)])]))

(define-syntax define-fluid
  (syntax-rules ()
    [(define-fluid x e)
     (set! *fluid-env*
       (cons (cons 'x e) *fluid-env*))]))

(define-syntax set-fluid!
  (syntax-rules ()
    [(set-fluid! x e)
     (cond [(assq 'x *fluid-env*)
            => (lambda (c)
                 (set-cdr! c e))]
           [else (error 'undefined-fluid 'x)])]))
```

This definition fails in the presence of *call/cc*, because a call to a continuation does not restore the fluid environment to the value it had at the capture of the continuation. A simple cob-based rewrite of *call/cc* takes care of this:

```
(define call/cc-f
  (let ([call/cc-orig call/cc])
    (lambda (proc)
      (call/cc-orig
        (lambda (k)
          (let* ([my-fluid-env *fluid-env*]
                 [cob (lambda (v)
                        (set! *fluid-env* my-fluid-env)
                        (k v))])
            (proc cob)))))))
```

Note that once we've defined the new fluids-aware *call/cc* variant, it's a good idea to reuse the *call/cc* name to refer to the variant. In essence, we retire the original *call/cc* from further use, as it would interfere with the correct functioning of the new operator. In standard Scheme, one could do this by simply re-**set!**ing the *call/cc* name, but this has problems as programs scale. In a Scheme with a module system [4], a more robust method is to define a new module that provides the new control operator under the *call/cc* name.

3 Unwind-protect and cobs

Friedman and Haynes already use cobs to tackle the problem of defining an unwind-protect (and the corresponding *call/cc* variant) for Scheme, and observe that there is a choice of meaningful unwind-protect semantics — the choice as they see it lying in the method of identifying which postludes to perform based purely on their position relative to the continuation nodes in the control tree.

However, automatic detection of the relevant postludes does not necessarily match user expectations. Sometimes it may be more suitable to allow the user explicitly specify which continuations, or continuation calls, ought to trigger unwind-protect postludes, as Kent Pitman [9] proposes. He suggests that *call/cc* as provided in the Scheme standard may be fundamentally misdesigned as it thwarts the creation of a pragmatic unwind-protect facility, and that it be replaced by one of two *call/cc*-like operators that he describes as more unwind-protect-friendly, while still providing full continuations.

Fortunately, the cob technique can implement both the Pitman variants, as we shall show in sections 4 and 5. Thus, at least as far as unwind-protect is concerned, Scheme's design does not pose a disadvantage. Indeed, given the variety of unwind-protect styles that are possible for Scheme (it's unlikely that the Friedman–Haynes and Pitman styles exhaust this list), learning the cob technique as a reliable and flexible way to implement the styles may be a better approach than enshrining one of the styles permanently in the standard.

3.1 call/cc-e

Pitman's first *call/cc* variant, which we will call *call/cc-e*, takes an additional argument whose boolean value decides if the captured continuation should be an *escaping* or a *full* continuation. Escaping continuations cannot be used to re-enter an already exited dynamic context, whereas full continuations have no such limitation. Thus, in the expressions:

```
(call/cc-e #t M)
(call/cc-e #f N)
```

M is called with an escaping continuation, whereas *N* is called with a full continuation.

In a Scheme with *call/cc-e*, for the expression (**unwind-protect** *B P*), the postlude *P* is run only if (1) *B* exits normally, or (2) *B* calls an escaping continuation that was captured outside the **unwind-protect** expression's dynamic extent.

3.2 call/cc-l

Pitman's second *call/cc* variant, which we will call *call/cc-l*, produces continuations which take an additional argument that decides

if that continuation call is to be the *last use* of that continuation. Thus, when evaluating the following expressions:

```
(define r 'to-be-set-below)

(call/cc-l
   (lambda (k)
      (set! r k)))

(r #f 'first-use-works)
(r #f 'second-use-works)
(r #t 'third-use-works-and-is-last-use)
(r #f 'fourth-attempted-use-will-not-work)
```

the fourth attempted use of the continuation *r* will error.

In a Scheme with *call/cc-l*, for the expression (**unwind-protect** *B P*), the postlude *P* is run only if (1) *B* exits normally, or (2) *B* calls a continuation for the (user-specified) last time, and that continuation does not represent a context that is dynamically enclosed by the **unwind-protect** expression.

In short, for *call/cc-e*, postludes are triggered only by continuations specified by the user to be escaping; and for *call/cc-l*, they are triggered only by continuation calls specified by the user to be their last use.

We will now use the cob technique to define each of these *call/cc* variants, and its corresponding unwind-protect, from *call/cc-f* (which we defined in section 2 from the primitive *call/cc*, also using a cob).

4 Unwind-protect that recognizes only escaping continuations

call/cc-e (unlike the standard *call/cc*) takes two arguments: The second argument is the procedure that is applied to the current continuation. Whether this continuation is an *escaping* or a *full* continuation depends on whether the first argument is true or false.

We implement *call/cc-e* by applying its second argument (the procedure) to a cob created using *call/cc-f*. The cobs created for *call/cc-e* #t (escaping continuations) and *call/cc-e* #f (full continuations) are different.

unwind-protect interacts with *call/cc-e* as follows: If the body is exited by a escaping continuation provided by a dynamically enclosing *call/cc-e* #t, then the postlude is performed. The postlude is *not* performed by full continuations or by escaping continuations created by a *call/cc-e* #t within the **unwind-protect**. To accomplish this, the cob generated by *call/cc-e* #t keeps a list (*my-postludes*) of all the postludes within its dynamic extent. Since a call to *call/cc-e* #t cannot know of the **unwind-protect**s that will be called in its dynamic extent, it is the job of each **unwind-protect** to update the *my-postludes* of its enclosing *call/cc-e* #ts. To allow the **unwind-protect** to access its enclosing *call/cc-e* #t, the latter records its cob in a fluid variable **curr-call/cc-cob**.

The following is the entire code for *call/cc-e* and its *unwind-protect-proc*, a procedural form of unwind-protect:

```
(define call/cc-e #f)
(define unwind-protect-proc #f)
(define-fluid ∗curr-call/cc-cob∗
   (lambda (v) (lambda (x) #f)))
(define-fluid ∗curr-u-p-alive?∗ (lambda () #t))

(let ([update (list 'update)]
      [delete (list 'delete)])

   (set! call/cc-e
      (lambda (once? proc)
         (if once?
            (call/cc-f
               (lambda (k)
                  (let∗
                     ([cob (fluid ∗curr-call/cc-cob∗)]
                      [my-postludes '()]
                      [already-used? #f]
                      [cob
                         (lambda (v)
                            (cond
                               [(eq? v update)
                                (lambda (pl)
                                   (set! my-postludes
                                      (cons pl my-postludes))
                                   ((cob update) pl))]
                               [(eq? v delete)
                                (lambda (pl)
                                   (set! my-postludes
                                      (delq! pl my-postludes))
                                   ((cob delete) pl))]
                               [already-used?
                                  (error 'dead-continuation)]
                               [else
                                  (set! already-used? #t)
                                  (for-each
                                     (lambda (pl) (pl))
                                     my-postludes)
                                  (k v)]))])
                     (let-fluid ([∗curr-call/cc-cob∗ cob])
                        (cob (proc cob))))))
               (call/cc-f
                  (lambda (k)
                     (let∗
                        ([my-u-p-alive? (fluid ∗curr-u-p-alive?∗)]
                         [cob
                            (lambda (v)
                               (if (my-u-p-alive?)
                                  (k v)
                                  (error 'dead-unwind-protect)))])
                        (cob (proc cob))))))))))

   (set! unwind-protect-proc
      (lambda (body postlude)
         (let ([curr-call/cc-cob (fluid ∗curr-call/cc-cob∗)]
               [alive? #t])
            (let-fluid ([∗curr-u-p-alive?∗ (lambda () alive?)])
               (letrec ([pl (lambda ()
                              (set! alive? #f)
                              (postlude)
                              ((curr-call/cc-cob delete) pl))])
                  ((curr-call/cc-cob update) pl)
                  (let ([res (body)])
                     (pl)
                     res)))))))
)
```

As we can see, the cob employed by *call/cc-e* #t (i.e., the part of the *call/cc-e* body that is active when its *once?* argument is true) is fairly involved. This is because, in addition to performing the jump, it has to respond to *update* and *delete* messages pertaining to its *my-postludes*. We have defined lexical variables *delete* and *update* so they are guaranteed to be different from any user values given to the cob. The cob also remembers its nearest enclosing cob (*prev-cob*), so that the *update* and *delete* messages can be propagated outward. (This is because *any* of the escaping continuations enclosing an **unwind-protect** can trigger the latter's postlude.) When the cob is called with a non-message, it performs all of its *my-postludes*, before calling its embedded continuation. It also remembers to set a local flag *already-used?*, because it is an error to call an escaping continuation more than once.[2]

The *call/cc-e* #f part, the one that produces full continuations, is fairly simple. Its cob simply remembers if its enclosing **unwind-protect** is alive, via the fluid variable **curr-u-p-alive?**. This is to prevent entry into an **unwind-protect** body that is known to have exited.

The corresponding *unwind-protect-proc* notes its nearest enclosing *call/cc-e* #t's cob, to let it know of its postlude. It also adds wrapper code to the postlude so that the latter can delete itself when it is done, and flag the **unwind-protect** as no longer alive. The body and the wrapped postlude are performed in sequence, with the body's result being returned.

The macro **unwind-protect** is defined as follows:

```
(define-syntax unwind-protect
  (syntax-rules ()
    [(unwind-protect body postlude)
     (unwind-protect-proc
       (lambda () body) (lambda () postlude))]))
```

The helper procedure *delq!* is used to delete a postlude from a list:

```
(define delq!
  (lambda (x s)
    (let loop ([s s])
      (cond [(null? s) s]
            [(eq? (car s) x) (loop (cdr s))]
            [else (set-cdr! s (loop (cdr s)))
                  s]))))
```

5 Unwind-protect that recognizes only last-use continuations

call/cc-l takes a single procedure argument (just like the standard *call/cc*), but the continuation it captures takes two arguments: The first argument, if true, marks that call as the *last use* of the continuation. The second argument is the usual transfer value of the continuation.

[2] Pitman's text calls the escaping continuations *single-use*, counting as a use the implicit use of the continuation (i.e., when the *call/cc-e* expression exits normally without explicitly calling its continuation). There are some design choices on what effect the use of such a continuation has on the use count of other continuations captured within its dynamic extent, whether they be single- or multi-use. For now, I assume there is no effect. If there were, such could be programmed by having the cob propagate kill messages to its nearest enclosed (not enclosing!) cob using fluid variables.

The corresponding unwind-protect's postlude is triggered by a continuation only on its last use.

call/cc-l, like *call/cc-e*, is implemented with a cob. (Unlike *call/cc-e*, *call/cc-l* does not create two types of continuations, so it doesn't need two types of cobs.) The *call/cc-l* cob looks very much like the union of the cobs for *call/cc-e*, except of course that whereas the *call/cc-e* triggers postludes for escaping continuations, the *call/cc-l* cob triggers them for continuations on their last use. Another difference is that the *call/cc-l* cob takes *two* arguments, like the user continuation it stands for. We use the cob's first argument for the message, which can be *update* and *delete* for manipulating the postludes, #f for marking non-last use, and any other value for last use.

As in the *call/cc-e* case, the cob is available as the fluid variable **curr-call/cc-cob** to an enclosed **unwind-protect**; and **unwind-protect** has a fluid variable **curr-u-p-alive?** so continuations can check it to avoid re-entering an exited **unwind-protect**. But we also associate another fluid variable with **unwind-protect**, viz., **curr-u-p-local-conts** — this is to keep track of continuations that were captured within the **unwind-protect**, for we view the call of a continuation whose capture and invocation are both local to the **unwind-protect** as non-exiting, and thus not worthy of triggering the postlude, even if it happens to be last-use. Each *call/cc-l* updates its enclosing **curr-u-p-local-conts**, and its cob's last call checks its current **curr-u-p-local-conts** before triggering postludes.

```
(define call/cc-l #f)

(define-fluid ∗curr-call/cc-cob∗ (lambda (b v) #f))
(define-fluid ∗curr-u-p-local-conts∗ '())
```

The following replaces (**set!** *call/cc-e* ...) in the code in section 4:

```
(set! call/cc-l
  (lambda (proc)
    (call/cc-f
      (lambda (k)
        (set-fluid! ∗curr-u-p-local-conts∗
          (cons k (fluid ∗curr-u-p-local-conts∗)))
        (let∗
          ([prev-cob (fluid ∗curr-call/cc-cob∗)]
           [my-u-p-alive?  (fluid ∗curr-u-p-alive?∗)]
           [my-postludes '()]
           [already-used? #f]
           [cob
             (lambda (msg v)
               (cond
                 [(eq? msg update)
                  (set! my-postludes (cons v my-postludes))
                  (prev-cob update v)]
                 [(eq? msg delete)
                  (set! my-postludes (delq! v my-postludes))
                  (prev-cob delete v)]
                 [already-used?
                   (error 'dead-continuation)]
                 [(not (my-u-p-alive?))
                  (error 'dead-unwind-protect)]
                 [msg
                   (set! already-used? #t)
                   (if (not
                         (memq
                           k
                           (fluid ∗curr-u-p-local-conts∗)))
```

```
                    (for-each (lambda (pl) (pl))
                              my-postludes))
                (k v)]
              [else (k v)])])
        (let-fluid ([*curr-call/cc-cob* cob])
          (cob #f (proc cob))))))))
```

The following replaces (**set!** *unwind-protect-proc* ...) in the code in section 4:

```
(set! unwind-protect-proc
  (lambda (body postlude)
    (let ([curr-call/cc-cob (fluid *curr-call/cc-cob*)]
          [alive? #t])
      (let-fluid ([*curr-u-p-alive?* (lambda () alive?)]
                  [*curr-u-p-local-conts* '()])
        (letrec ([pl (lambda ()
                       (set! alive? #f)
                       (postlude)
                       (curr-call/cc-cob delete pl))])
          (curr-call/cc-cob update pl)
          (let ([res (body)])
            (pl)
            res))))))
```

The only significant difference between this *unwind-protect-proc* and the one in section 4 is that it initializes the fluid variable **curr-u-p-local-conts**, which dynamically enclosed calls to *call/cc-l* can update.

6 Conclusion

We see that the Friedman-Haynes cob technique for deriving constrained forms of *call/cc* is a reliable way to implement various forms of unwind-protect — both the Pitman-style ones that rely on explicit user annotation, and the Friedman-Haynes ones that depend on the relative positions of unwind-protects and continuations on the control tree.

The cob technique is not the only way to derive unwind-protect — for an ingenious way to derive *call/cc-e* using Scheme's built-in *dynamic-wind*, see Clinger [2]. However, the cob approach remains a predictable workhorse for systematically experimenting with new styles and modifying existing styles. This kind of flexibility is especially valuable for unwind-protect since the latter cannot have a canonical, once-and-for-all specification in Scheme, making it important to allow for multiple library solutions.

7 Acknowledgments

I thank Matthias Felleisen and Robert Bruce Findler for helpful discussions.

8 References

[1] Bruce F. Duba, Robert Harper, and Dave MacQueen, "Typing First-Class Continuations in ML", in *18th ACM Symp. on Principles of Programming Languages*, pp. 163–173, 1991.

[2] William Clinger, `http://www.ccs.neu.edu/home/will/Temp/uwesc.sch`.

[3] Matthias Felleisen, "On the Expressive Power of Programming Languages", in *European Symp. on Programming*, 1990, pp. 134–151.

[4] Matthew Flatt, "Composable and Compilable Macros: You Want It *When?*", in *International Conf. on Functional Programming*, 2002.

[5] Daniel P. Friedman and Christopher T. Haynes, "Constraining Control", in *12th ACM Symp. on Principles of Programming Languages*, 1985, pp. 245–254.

[6] Christopher T. Haynes and Daniel P. Friedman, "Engines Build Process Abstractions", in *ACM Symp. on Lisp and Functional Programming*, 1984, pp. 18–24.

[7] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, "Continuations and Coroutines", in *ACM Symp. Lisp and Functional Programming*, 1984, pp. 293–298.

[8] Richard Kelsey and William Clinger (eds.), *Revised^5 Report on the Algorithmic Language Scheme ("R5RS")*, `http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs.html`, 1998.

[9] Kent Pitman, *UNWIND-PROTECT vs Continuations*, `http://www.nhplace.com/kent/PFAQ/unwind-protect-vs-continuatins.html`.

[10] Dorai Sitaram, *Teach Yourself Scheme in Fixnum Days*, `http://www.ccs.neu.edu/~dorai/t-y-scheme/t-y-scheme.html`.

Enabling Complex User Interfaces In Web Applications With send/suspend/dispatch

Peter Walton Hopkins
Computer Science Department
Brown University
Box 1910, Providence, RI 02912
phopkins@cs.brown.edu

ABSTRACT

Web application programmers who use PLT Scheme's web server enjoy enormous benefits from the **send/suspend** primitive, which allows them to code in a direct style instead of in continuation-passing style as required by traditional CGI programming. Nevertheless, **send/suspend** has limitations that hinder complex web applications with the rich interfaces expected by users. This Scheme Pearl introduces a technique for "embedding" Scheme code in URLs and shows how this facilitates developing complex web-based user interfaces.

1. USER INTERFACES IN HTML

As web applications become more popular and powerful, the demands on their interfaces increase. Users expect complex interface elements that emulate those found in desktop applications: tabs for switching among screens of data, tables that can sort with clicks on their column headers, confirmation "dialogs," etc.

The screenshot in Figure 1 shows a web application with two such UI elements: a row of tabs across the top and a table with clickable column headers. The page is from CONTINUE [4], a Scheme web application for accepting paper submissions and managing the conference review process. I re-wrote CONTINUE using the technique in this paper, so I will use it as a recurring example of a web application with a rich user interface.

HTML provides several user interface elements (radio buttons, check boxes, text fields, etc.) and web browsers give them an OS-appropriate appearance and behavior. Interface elements not provided by HTML—tabs, for example—must get their appearance from HTML mark-up and their behavior from the web application code. The web application programmer must devote a non-trivial amount of code to emulating complex elements by reducing them to HTML's universal interface element: the hyperlink.

In the screen shot, each tab is a link, each column header is a link, and each paper title is a link. When the user clicks on any of these the browser will send a request to the servlet and the servlet must respond with a new web page that has the appropriate change in interface. This means that the links must be constructed—and the servlet must be coded to recognize the construction—in a way that can convey their meaning. Is it a click on a tab? Or should the columns be re-sorted? Or should an entirely different page (a CONTINUE example is a page showing reviews for a paper) be returned?

The page from CONTINUE shows three very different classes of links: tabs, column headers, and paper titles. This diversity of hyperlinks makes implementing this page with the PLT web server's core primitive, **send/suspend**, difficult. This paper describes an extension to **send/suspend**, called **send/suspend/dispatch**, that vastly simplifies the code necessary for complex pages.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Peter Walton Hopkins.
This paper was partially supported by NSF Grants ESI-0010064, ITR-0218973, SEL-0305949 and SEL-0305950, and by Brown University's Karen T. Romer UTRA program.

2. THE PLT WEB SERVER

Because of the nature of CGI—the web application process halts after returning a page to the browser—"traditional" web programming must be written in a continuation-passing style [5, 2]: a web page sent to the browser must contain enough data (commonly in hidden form fields) so that the server can pick up where it left off when it had to terminate.

The PLT web server [3] uses Scheme's first-class continuations to avert a CPS transformation and the problems (unclear program flow, serialization of data into strings, exposure of some application internals) associated with it.

send/suspend is the PLT Scheme web server's primitive for capturing a servlet's continuation. It consumes a page-generating function of one argument: a URL that will resume the continuation, which by convention is named *k-url*. The result of evaluating the page-generating function with a *k-url* is sent to the user's web browser. When a link to *k-url* is clicked the browser makes a request to the servlet and **send/suspend** resumes the continuation by passing it the request.

send/suspend will only capture one continuation per page: the "actual" continuation waiting for the browser's request. But, most web application pages have multiple "logical" continuations pending. The CONTINUE page in Figure 1 has three: one waiting for a tab to switch to, one waiting to sort the list, and one waiting for a paper to show in detail. With **send/suspend**, the programmer must shoehorn a page's logical continuations into the one actual continuation that will be resumed. The code to do this dispatching is both fragile and generalizes poorly.

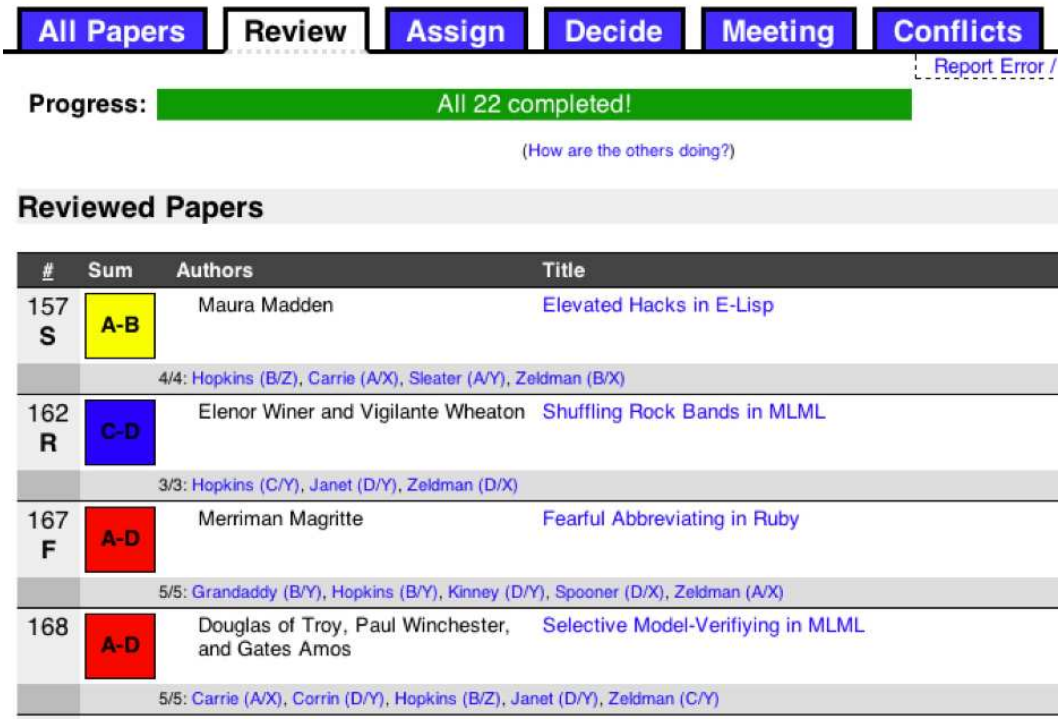


Figure 1: Screenshot from the CONTINUE server

3. DISPATCHING WITH SEND/SUSPEND

In general, **send/suspend** requires the programmer to encode enough data into a page's hyperlinks that the actual continuation can dispatch to the correct logical continuation. HTTP has at least two methods of adding data to a URL, the most flexible of which is to append a query string of the form ?key_1=$value_1$&key_2=.... The code to do this for the page titles in the CONTINUE example might look like:

```
1 = (send/suspend
       (lambda (k-url)
         `(html ...
           (a ([href ,(format "~a?paper=157" k-url)])
              "Elevated Hacks in E-Lisp")
           (a ([href ,(format "~a?paper=162" k-url)])
              "Shuffling Rock Bands in MLML")
           (a ([href ,(format "~a?paper=167" k-url)])
              "Fearful Abbreviating in Ruby"))))
```

Or, more realistically, with a map over a list:

```
2 = (send/suspend
       (lambda (k-url)
         `(html ...
           ,@(map
              (lambda (paper-num)
                `(a ([href ,(format "~a?paper=~a"
                                    k-url paper-num)])
                    ,(paper-title paper-num)))
              (get-all-papers)))))
```

This encoding solution was proven workable by the first few versions of CONTINUE. All it needs is code to interpret the request from the browser, pick out which paper the user clicked on, and dispatch accordingly:

```
(define (show-list/review)
  (let* ([request 2]
         [bindings (request-bindings request)]
         [paper-num
           (string→number
             (car (extract-bindings 'paper bindings)))])
    (show-paper paper-num)))
```

The servlet environment handily converts the URL's query string into a list of (*key* . *value*) pairs, and it also provides the *extract-binding* function to return all values matching a given key. A call to the *show-paper* function sends the requested paper's page to the user's browser.

send/suspend is perfectly adequate in this situation because there is only one logical continuation to resume: the one waiting for a paper to display. Adding just one additional logical continuation—for example, one that is waiting to switch to a particular tab—will make the above dispatching code much more complicated.

First, here is the HTML-generating code for the page with tabs. URLs are now postfixed by either a `?paper=`n or a `?tab=`n. The class attribute is used to signal that the Review tab is current and should be displayed differently.

```
3 = (send/suspend
       (lambda (k-url)
         `(html ...
           (ul ([id "tabs"])
               (li (a ([href ,(format "~a?tab=1" k-url)])
                      "All Papers"))
               (li (a ([href ,(format "~a?tab=2" k-url)]
                       [class "selected"])
                      "Review"))
               (li (a ([href ,(format "~a?tab=3" k-url)])
                      "Assign"))
               (li (a ([href ,(format "~a?tab=4" k-url)])
                      "Decide")))
           ...
           ,@(map
                (lambda (paper-num)
                  `(a ([href ,(format "~a?paper=~a"
                                      k-url paper-num)])
                      ,(paper-title paper-num)))
                (get-all-papers)))))
```

The dispatching code must now check which logical continuation is being resumed: tab or paper.

```
(define (show-list/review)
  (let* ([request 3]
         [bindings (request-bindings request)])

    (cond [(pair? (extract-bindings 'paper bindings))
           (let ([paper-num
                  (string→number
                    (car (extract-bindings 'paper bindings)))])
             (show-paper paper-num))]

          [(pair? (extract-bindings 'tab bindings))
           (let ([tab-num
                  (string→number
                    (car (extract-bindings 'tab bindings)))])
             (case tab-num
               [(1) (show-list/all)]
               [(2) (show-list/review)]
               [(3) (show-list/assign)]
               [(4) (show-list/decide)]))])))
```

This dispatching code is more involved than the code for the previous case, though extending it for additional logical continuations is straightforward: the programmer adds more HTML to encode more data in a URL, then he adds another clause to the **cond** to handle the new logical continuation. But, though workable, there are two major problems with this pattern of web programming.

The first is that the code to generate the HTML and encode data into the URLs is separate from the code to decode the URLs and perform the dispatching. These pieces of code are tightly dependent, so changes to one must be matched by changes to the other. Their separation imposes a higher maintenance burden to ensure that they are kept in synch.

The second problem is that the programmer cannot easily generalize complex interface elements into their own functions. Though not very not evident from a single example, this is a major limitation for medium-to-large–scale web applications because it breaks down abstractions and forces code duplication.

For example, CONTINUE uses tabs on nearly every page, so a function to create them and handle Their behavior would be useful from a development and a maintenance standpoint. But, **send/suspend** makes such a function impractical. The closest solution is one function that generalizes the HTML and another the behavior, with the understanding that these two must be kept closely in synch:

```
(define generate-tabs (k-url tab-list selected)
  `(ul ([id "tabs"])
    ,@(map
       (lambda (tab-pair)
         `(li (a ([href ,(format "~a?tab=~a"
                                 k-url (car tab-pair))]
                  [class ,(if (equal? (car tab-pair) selected)
                              "selected"
                              "")])
                 ,(car tab-pair))))
       tab-list)))

(define dispatch-tabs (request tab-list)
  (let* ([bindings (request-bindings request)]
         [tab-bindings (extract-bindings 'tab bindings)])
    (and (pair? tab-bindings)
         ((cdr (assoc (car tab-bindings) tab-list))))))

(define (show-list/review)
  (let* ([tab-list`(["All" . ,show-list/all]
                    ["Review" . ,show-list/review]
                    ["Assign" . ,show-list/assign]
                    ["Decide" . ,show-list/decide])]
         [request 4]
         [bindings (request-bindings request)])

    (cond [(pair? (extract-bindings 'paper bindings))
           ... ]
          [(dispatch-tabs request tab-list)])))

4 = (send/suspend
       (lambda (k-url)
         `(html ...
           ,(generate-tabs k-url tab-list "Review")
           ...
           ,@(map
                (lambda (paper-num)
                  `(a ([href ,(format "~a?paper=~a"
                                      k-url paper-num)])
                      ,(paper-title paper-num)))
                (get-all-papers)))))
```

Even with this generalized version, the page function is still responsible for dispatching each logical continuation. An interface element cannot be added to a page without both a clause in the dispatch code to handle it and all the necessary dispatching data (in this case, *tab-list*) in scope. With **send/suspend** there is no clear way to have a single, opaque function that adds a set of tabs to a page.

4. SOLUTION: SEND/SUSPEND/DISPATCH

send/suspend/dispatch solves the above two problems by allowing the programmer to specify a continuation, in the form of a closure, for every URL on a web page. This generalizes the encoding,

decoding, and dispatch necessary with **send/suspend**'s single continuation model. By "embedding" closures into the page the programmer can easily write complex UI elements because presentation and behavior are local in the file and generalization of elements into functions is possible.

send/suspend/dispatch presents an interface very similar to **send/suspend**'s: pages are still generated by a function of one argument. But instead of the static *k-url*, page-generating functions receive a function—conventionally called *embed/url*—that consumes a function of one argument and returns a unique URL. The page-generating function uses *embed/url* to embed continuations into URLs. When a URL is requested by the browser the continuation embedded in that URL will be resumed, receiving the browser's request as its argument.

The following is the first example from above, converted to use **send/suspend/dispatch**. A separate dispatching step is no longer necessary; **send/suspend/dispatch** manages resuming the continuation when a URL is accessed.

```
(define (show-list/review)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        ,@(map
           (lambda (paper-num)
             `(a ([href ,(embed/url
                          (lambda _
                            (show-paper paper-num)))])
                 ,(paper-title paper-num)))
           (get-all-papers))))))
```

Tabs are added easily to the above code, again with no explicit dispatching:

```
(define (show-list/review)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        (ul ([id "tabs"])
          (li (a ([href ,(embed/url
                          (lambda _ (show-list/all)))])
                 "All Papers"))
          (li (a ([href ,(embed/url
                          (lambda _ (show-list/review)))]
                  [class "selected"])
                 "Review"))
          ...)
        ...
        ,@(map
           (lambda (paper-num)
             `(a ([href ,(embed/url
                          (lambda _
                            (show-paper paper-num)))])
                 ,(paper-title paper-num)))
           (get-all-papers))))))
```

The above code examples demonstrate how presentation code and behavior code are local to each other in the file with **send/suspend/dispatch**. The flow of control in these examples is clear because of that locality.

Now we can construct a more useful generalization of tabs. Unlike the previous pair of functions, this function is self-contained: because all dispatching is handled by **send/suspend/dispatch**, the result of *generate-tabs* can simply be dropped into a page and its behavior will be handled properly.

```
(define generate-tabs (embed/url tab-list selected)
  `(ul ([id "tabs"])
    ,@(map
       (lambda (tab-pair)
         `(li (a ([href ,(embed/url
                          (lambda _ ((cdr tab-pair))))]
                  [class ,(if (equal? (car tab-pair) selected)
                              "selected"
                              "")])
                 ,(car tab-pair))))
       tab-list)))
```

The ability to write self-contained functions like *generate-tabs* is essential for adding complex UI elements to web applications. For example, CONTINUE has a general function, *make-paper-list*, to create lists of papers like the one in Figure 1. It can be added to the previous example:

```
(define (show-list/review view-info)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        ,(generate-tabs embed/url `(...) "Review")
        ,(make-paper-list
           (get-all-papers) embed/url show-paper
           show-list/review view-info)))))
```

make-paper-list takes a list of papers to show, the *embed/url* function, a function to invoke when a paper is clicked on, a callback to re-display the current page, and data defining how to show the list. To *show-list/review*, *view-info* is an opaque vehicle for passing data back into *make-paper-list* when the callback is used.

Clickable column headers are implemented entirely within *make-paper-list*: they call the callback (*show-list/review* in this case) with *view-info* changed to include the new sort. This will re-display the same page the user was looking at, but the sorting of the papers will be different.

make-paper-list can be extended with logical continuations for additional behaviors (filtering by rating, for example) without any changes to *show-list/review*, *show-list/assign*, or any other function that calls *make-paper-list*. This is possible because the *show-list/*∗ functions do not have to handle any dispatching themselves.

4.1 Is s/s/d A Step Backwards?

One of the most useful features of **send/suspend** is that it prevents a global CPS transformation of web application code. At first glance, **send/suspend/dispatch** looks regressive because it forces the programmer to be explicit with his continuations.

Though code using **send/suspend/dispatch** resembles CPS code, this is acceptable for several reasons. First, the CPS transformation is local, not global. Code is still written in a mostly direct style, and **send/suspend/dispatch**'s embedded continuations are arguably clearer than the separate dispatch **cond** from **send/suspend**

code. Second, in practice the embedded continuations tend to simply call named functions like in the tab example above. This also keeps the flow of control clear. Finally, a multiple-continuation model fits a web page more accurately than a single-continuation model. As the CONTINUE example shows, web pages often have several logical continuations active at one time, and **send/suspend/dispatch** correctly captures that pattern.

4.2 send/suspend/dispatch With Forms

In all previous examples the embedded continuations used the _ argument convention to ignore the value they were resumed with, which is the request data sent from the browser. When the continuation is embedded in a hyperlink the request data is rarely relevant, but it is necessary when the continuation is embedded into the action URL for a form:

```
`(form ([action ,(embed/url
                   (lambda (request)
                     (handle-form-bindings
                       (request-bindings request))))])
  ...
  (input ([type "submit"] [name "button"]
          [value "Save"]))
  (input ([type "submit"] [name "button"]
          [value "Save and Continue"])))
```

The bindings for a request contain the data from the form. A continuation embedded in a form will access these and process them as necessary.

An intrinsic shortcoming of HTML forms—not addressed by either **send/suspend** or **send/suspend/dispatch**—is that each form has a single action URL that form data is always sent to. This makes it impossible to distinguish between different submit buttons using URLs.

Some forms in the CONTINUE server faced this problem. For example, when assigning PC members to review a paper the PC chair has one button to save his decisions and remain looking at the same paper, and another button to save his decisions and automatically show another paper. These two buttons are in the same `form` tag because they need to share the same form elements (checkboxes, etc.). Because they share a form, they share an action URL and, therefore, a single re-entry point in the servlet. The code embedded in the form's action URL must handle the dispatch on which button was clicked:

```
(define (handle-form-bindings bindings)
  (let ([button (extract-binding/single 'button bindings)])
    (save-assignment-data request)
    (cond [(equal? button "Save")
           ;; show same paper
           ]
          [(equal? button "Save and Continue")
           ;; show new paper
           ])))
```

The *handle-form-bindings* function will receive all the data from the form. But it must fall back on **send/suspend**-like dispatching to determine if the user clicked "Save" or "Save and Continue."

If the programmer could specify unique URLs for each button he could use **send/suspend/dispatch** to embed separate functions for each button. With the current state of HTML the only solution to the two-button problem is to use a single URL and examine the request bindings to determine which button was clicked.

5. IMPLEMENTATION

send/suspend/dispatch is defined in terms of **send/suspend**, augmenting it by transparently handling the encoding of *k-url* and the subsequent dispatching. The code for **send/suspend/dispatch** is in Figure 2. Figure 3 contains the necessary helper functions.

First, **send/suspend/dispatch** creates a hash table (*embed-hash*) to store embedded functions. The keys for this hash table are random numbers and are generated by the *unique-hash-key* funcion.

1: **send/suspend/dispatch** calls **send/suspend** to send the page to the browser and get the response (what the user clicked on). *page-func* is the user's page-generating function, which takes *embed/url* as its argument.

When called with no arguments, *embed/url* just returns *k-url*. When called with a function to embed as its argument it uses 2 to generate a unique, random key and store the function (*embed/func*) in the hash table with that key. *embed/url* then calls *url-append/path* on *k-url* and the key to create a URL that will resume **send/suspend**'s actual continuation but carry with it an identifier for the logical continuation to resume.

A **send/suspend** URL (disregarding the `http://` and server) looks like this:

```
/servlets/cont.ss;id313*k2-1167813005
```

The text following the `;` identifies the continuation that **send/suspend** will resume. A **send/suspend/dispatch** URL includes the hash key in the path portion of the URL:

```
/servlets/cont.ss/34412;id313*k2-1167813005
```

Once the browser responds with a request, **send/suspend/dispatch** extracts the key from the URL by calling *post-servlet-path* to get the part of the path following the servlet's extension. The first piece of this path is the key.

3: Finally, **send/suspend/dispatch** looks up the key in the hash table (returning a simple error page if it is not found) to get the continuation embedded in the link the user clicked. It calls this function with the browser's request as an argument.

6. CONCLUSION

send/suspend/dispatch, an extension to the PLT web server's **send/suspend**, vastly simplifies servlet coding by enabling a very natural abstraction: that each URL on a web page is tied to a separate, pending continuation. **send/suspend**'s one-continuation model led to inappropriately divided code, prevented generalizations, and forced the programmer to handle low-level issues of URLs and query strings.

Web pages that had several logical continuations, such as those emulating complex user interface elements, become natural and straightforward to implement with **send/suspend/dispatch**.

```
(define/contract send/suspend/dispatch
  (page-func/ssd-contract . → . any)
  (lambda (page-func)
    (let* ([embed-hash (make-hash-table)]
           [request 1 ]
           [path
            (post-servlet-path
              (url→string (request-uri request)))])
      (if (null? path)
          request
          3 ))))
1 = (send/suspend
      (lambda (k-url)
        (page-func
          (case-lambda
            [() k-url]
            [(embed-func) 2 ]))))
2 = (let ([key (unique-hash-key embed-hash)])
      (hash-table-put! embed-hash key embed-func)
      (url-append/path k-url key))
3 = ((hash-table-get
      embed-hash
      (string→number (car path))
      (lambda ()
        (lambda _
          (send/back
            '("text/plain"
              "ERROR: Key was not found in "
              "send/suspend/dispatch hash table")))))
     request)
```

Figure 2: send/suspend/dispatch Implementation

7. ACKNOWLEDGMENTS

Thanks to Shriram Krishnamurthi for the original CONTINUE code and for invaluable help advising me as I rewrote it and put together this paper.

8. REFERENCES

[1] R. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[2] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *IEEE International Symposium on Automated Software Engineering*, pages 211–222, Nov. 2001.

[3] 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.

[4] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, 2003. **Invited Paper**.

[5] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

```
;; hashtable → number
;; given a hash table, finds a number not already used as a key
(define unique-hash-key
  (lambda (ht)
    (let ([key (random 200000)])
      (let/ec exit
        (hash-table-get ht key (lambda () (exit key)))
        (unique-hash-key ht)))))

;; string → listof string
;; takes a string that is the path portion of a URL
;; finds the path that follows the .ss extension, splits it
;; at / characters, and returns the list
(define post-servlet-path
  (lambda (s-url)
    (let ([result (regexp-match "\\.ss(/[^;#\\?]*)" s-url)])
      (if result
          (filter
            (lambda (s) (> (string-length s) 0))
            (regexp-split "/" (cadr result)))
          null))))

;; string any → string
;; adds a datum to the end of the path of a URL represented
;; as a string. The added datum comes before any query or
;; parameter parts of the URL.
(define url-append/path
  (lambda (s-url rel-path)
    (let ([url (string→url s-url)])
      (url→string
        (make-url
          (url-scheme url)
          (url-host url)
          (url-port url)
          (format "~a/~a" (url-path url) rel-path)
          (url-params url)
          (url-query url)
          (url-fragment url))))))

;; contracts for embed/url and page-generating functions,
;; for use with PLT Scheme's contracts [1]
(define embed/url-contract
  ((→ string?) . case→ .
   ((request? . → . any) . → . string?)
   (string? (request? . → . any) . → . string?)))
(define page-func/ssd-contract
  (embed/url-contract . → . any))
```

Figure 3: Helper functions for send/suspend/dispatch

Well-Shaped Macros

Ryan Culpepper, Matthias Felleisen
Northeastern University
Boston, MA 02115
Email: ryanc@ccs.neu.edu

Abstract

Scheme includes an easy-to-use and powerful macro mechanism for extending the programming language with new expression and definition forms. Using macros, a Scheme programmer can define a new notation for a specific problem domain and can then state algorithms in this language. Thus, Scheme programmers can formulate layers of abstraction whose expressive power greatly surpasses that of ordinary modules.

Unfortunately, Scheme's macros are also too powerful. The problem is that macro definitions extend the parser, a component of a language's environment that is always supposed to terminate and produce predictable results, and that they can thus turn the parser into a chaotic and unpredictable tool.

In this paper, we report on an experiment to tame the power of macros. Specifically, we introduce a system for specifying and restricting the class of shapes that a macro can transform. We dub the revised macro system `well-shaped macros`

1. MACROS ARE USEFUL

Over the past 20 years, the Scheme community has developed an expressive and useful standard macro system [8]. The macro system allows programmers to define a large variety of new expression and definition forms in a safe manner. It thus empowers them to follow the old Lisp maxim on problem-solving via language definition, which says that programmers should formulate an embedded programming language for the problem domain and that they should express their solution for the domain in this new language.

Standard Scheme macros are easy and relatively safe to use. To introduce a macro, a programmer simply writes down a rewriting rule between two syntactic patterns [10], also called `pattern`and `template` Collectively the rules specify how the macro expander, which is a component of the parser, must translate surface syntax into core Scheme, that is, Scheme without any extensions. Specifically, the (left-hand side) pattern specifies those S-expressions that the expander should eliminate in favor of the (right-hand side) template. Furthermore, the expander is "hygienic" [9] and "referentially transparent" [3], which means that macro expansion automatically respects the lexical scope of the program.

It is a key characteristic of Scheme macros that their uses are indistinguishable from built-in forms. As for built-in and defined functions, a programmer should not, and without context cannot, recognize whether a form is introduced via a macro or exists in core Scheme. Due to this uniformity, (teams of) programmers can build many-tiered towers of abstraction, each using conventional procedural libraries as well as new linguistic mechanisms.[1]

Although the Scheme authors have clearly tamed Lisp's programmed macros and C's string rewriting macros, they have still left the macro sublanguage with as much power as the untyped lambda calculus. In particular, macro expansion can create ill-formed core syntax, and it can diverge. We illustrate this point with some examples in the next section.

The situation suggests that we study ways of taming Scheme macros with a type system.[2] In this paper, we report on the results of one such an experiment. In section 2 we explain how Scheme macros can still perform undesirable computations. In sections 3 and 4, we introduce a modified macro system that allows a Scheme implementation to determine whether macro definitions and programs are syntactically well-formed. In section 5, we compare our work to related work and propose some future research.

2. MACROS ARE TOO POWERFUL

Standard Scheme macros suffer from two problems. On one hand, they can turn the macro expander into an infinite loop. Since the expander is a part of the parser, a programmer can turn the most reliable part of an ordinary programming environment into a useless tool. On the other hand, a macro can misapply Scheme's syntactic constructors, creat-

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Ryan Culpepper, Matthias Felleisen.

[1]We readily acknowledge that building such towers poses additional, serious problems for language designers [6, 11], but this topic is beyond the scope of our paper.

[2]If we were to eliminate ellipses and introduce an induction schema, our result would literally reconstruct for macro systems what the type discipline of Church did for the original lambda calculus.

ing S-expressions that even Scheme cannot interpret.

Consider the following simple macro definition:

```
(define-syntax diverge
  (syntax-rules () ((_) (diverge))))
```

It introduces a macro that replaces occurrences of (`diverge`) with itself, thus causing the extended parser to diverge. This example of bad behavior is trivial, however, compared to the full power of macros. It is a simple exercise to write a set of macros that simulate a pushdown automaton with two stacks.

While the introduction of unbridled computational power is a problem, we are truly concerned with macros that create ungrammatical Scheme expressions and definitions. Macro definitions can go wrong in two ways. First, the user of a macro may use it on proto-syntactic forms that the creator of the macro didn't anticipate. Consider an increment macro:

```
(define-syntax ++
  (syntax-rules ()
    ((_ x) (begin (set! x (+ x 1)) x))))
```

Furthermore consider the following (ab)use of the macro:

```
... (++ (vector-ref a 0)) ...
```

Clearly, the creator of the macro didn't expect anyone to use the macro with anything but an identifier, yet the user—perhaps someone used to a different syntax—applied it to a vector-dereferencing expression.

Second, the macro creator may make a mistake and abuse a syntactic construction:

```
(define-syntax where
  (syntax-rules (is)
    ((_ bdy lhs is rhs) (let ([rhs lhs]) bdy))))
```

Here the intention is to define a `where` macro, which could be used like this:

```
(where (+ x 1) y is 5)
```

Unfortunately, the right-hand side of the rewriting rule for `where` abuses the `rhs` pattern variable as a `let`-bound identifier and thus creates an ill-formed expression.

At first glance, the situation is seemingly analogous to that of applying a programmer-defined Scheme function outside of its intended domain or applying an erroneous function. In either case, the programmer receives an error message and needs to find the bug. The difference is, however, that many Scheme systems report the location of a safety violation for a run-time error and often allow the programmer to inspect the stack, which provides even more insight. In Chez Scheme [4], for example, the (ab)user of `++` receives the report that the syntax

```
(set! (++ (vector-ref v 0)) (+ (++ (...)) 1))
```

is invalid; the user of `where` finds out that

```
(let ((5 x)) (+ x 1))
```

is invalid syntax, without any clue of which portion of the program introduced this bug. Even in DrScheme [5], a sophisticated IDE that employs source code tracing and high-lighting to provide visual clues, a programmer receives difficult-to-decipher error messages. The (ab)use of `++` macro highlights the `vector` dereference and reports that some `set!` expression is ill-formed, which at least suggests that the error is in the use of `++`. In contrast, for the use of `where`, DrScheme highlights the `5` and suggests that `let` expected an identifier instead. This leaves the programmer with at most a hint that the macro definition contains an error. In this paper, we outline the design and implementation of a tool for catching these kinds of mistakes.

3. CONSTRAINING MACROS BY SHAPES

One way to tame the power of Scheme macros is to provide a type system that discovers errors before an implementation expands macros. In this section, we present such a type system, dubbed a `shape`system. The primary purpose of the type system is to assist macro programmers with the disovery of errors inside of macro definitions, but we also imagine that the users of macro libraries can employ the system to inspect their macro uses.

In the first subsection, we present the language of our model. In the second and third section, we gradually introduce the system of shapes. In the fourth section, we explain why a conventional type checking approach doesn't work. In the last subsection, we sketch the principles of our approach; the actual implementation is described in the next section.

3.1 The language

Figure 1 specifies the programming language of our model: the surface syntax and the core syntax. The surface syntax consists of a core syntax plus macro applications. The core syntax consists of definitions and expressions. The underlined portions of the figure indicate the parts of the language that belong to the surface syntax but not core syntax.

A program is a sequence of macro definitions followed by a sequence of forms in the surface syntax. Macro definitions use `syntax-laws`, a variant of Scheme's `syntax-rules`. More specifically, `syntax-laws` is a version of `syntax-rules` that accommodates shape annotations.

One restrictions of our model is that the set of primitive keywords, macro keywords, identifiers, and pattern variables are assumed to be disjoint subsets of Scheme's set of tags. This eliminates the possibility of `lambda`-bound variables shadowing global macros.

```
program   ::= macro-def* top-level*
top-level ::= def | expr
def       ::= (define id expr) | (macro . s-expr)
expr      ::= id | number | (expr expr*)
            | (lambda (id*) expr)
            | (lambda (id* . id) expr)
            | (quote s-expr) | (macro s-expr)
macro-def ::= (define-syntax macro
                (syntax-laws etype
                  ((_ . pattern) guards s-expr)*))
pattern   ::= pvar | () | (pattern pattern)
            | (pattern . ())
guards    ::= ((pvar stype)*)
tag       ::= unspecified countable set
keyword   ::= lambda | define | quote
            | define-syntax | syntax-laws | ...
id        ::= subset of tag disjoint from macro
pvar      ::= subset of tag disjoint from macro, id
s-expr    ::= keyword | macro | id | pvar
            | number | () | (s-expr s-expr)
```

$$(x_1 . (x_2 . \ldots . ())) \equiv (x_1 \ldots x_n)$$

Figure 1: Syntax

3.2 Base types and shape types

A close look at the grammar in figure 1 suggests that a macro programmer should know about four base types:

1. expression, which denotes the set of all core Scheme expressions;
2. definition, which denotes the set of all core Scheme definitions;
3. identifier, which denotes the set of all lexical identifiers; and
4. any, which denotes the set of all S-expressions.

The first three correspond to the basic syntactic categories of an ordinary Scheme program. The separation of identifiers from expressions is important so that we can deal with syntactic constructors such as `lambda` and `set!` which require identifiers in specific positions rather than arbitrary expressions. Scheme's quoting sublanguage also requires the introduction of a distinguished type any so that we can describe the set of all arguments for `quote`.

The four base types are obviously related. Once we classify a tag as identifier, we can also use it as an expression and in a `quote` context. Similarly, a definition can also occur in a quoted context, but it cannot occur in lieu of an expression. Figure 2 summarizes the relationship between the four base types.

At first glance, the collection of base type may suffice to describe the type of a macro. As we know from the Scheme report, a macro's output is always an expression or a definition. This explains the specification of `etype` which is the collection of range types for macro definitions. Concerning a macro's inputs, however, the Scheme report makes no restrictions. Following the precedence of type theory (for functional programming), we start with the idea that a macro programmer should specify types of the formal parameters, which are the pattern variables in each clause.

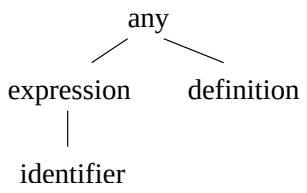


Figure 2: Base types

Take a look at this example:

```
(define-syntax simple-let
  (syntax-laws expression
    ((_ (var expr) body)
     ((var identifier)
      (expr expression)
      (body expression))
     ((lambda (var) body) expr))))
```

The macro definition introduce a simple form of `let` that binds one identifier to the value of one expression in some second expression. The (left-hand side) pattern therefore includes three pattern variables whose types are specified in the guard of our `syntax-laws` form. Still, it would be misleading to say that `simple-let` has a type like

identifier expression expression → expression

because that would completely ignore that the macro use must group the `var` and the `expr` components so that they are visually distinct from `body`.

Put more generally, a macro programmer specifies the general shape of a macro's input with two components: the types of the pattern variables and the pattern of grouping parentheses. Since checking the use of macros is about checking the well-formedness of its subexpressions, the types for macros must take these parentheses into account. Based on these observations, we introduce `shape types` or `shapes` for short, to describe the structure of the terms that macros consume. Shape types include the base types and construct compound types using pair types, the `null` type, case types, and arrow types. The latter two are only useful to describe an entire macro; we do not deal with macros as arguments in this paper.

Using shape types, we can specify the type of `simple-let` as

((identifier . (expression . ())) . (expression . ())) → expression

```
etype := expression | definition
btype := etype | identifier | any
stype := btype | () | (stype stype)
       | stype → stype | (case stype*)
       | (stype ... stype)
```

Figure 3: Types

The choice of pairing as the basic constructor in the model represents our assumption that a macro application is a pair of the macro keyword and some S-expression.

3.3 Sequences

The type language described so far is not rich enough to describe macros like `and` and `let` and primitive syntax like `lambda`. These macros employ ellipses, because the core of Scheme allows programmers to write down arbitrarily long sequences of expressions without intervening visual markers. Since ellipses are an integral part of the pattern and template language of macro definitions, we extend our type language with the sequence shape type constructor.

Ellipses occur in two radically different ways in macros. In patterns, an ellipsis must always end a list. That is, `(a ...)` is a valid pattern, but `(a ... b)` is not. In templates, an ellipsis may be followed by any template. Thus, `(a ... b)` and `(a ... . b)` are both valid templates. To cover both cases, our shape type constructor for sequences handles the general case for templates. Ellipses in patterns are described by sequences whose final part is always `()`. Figure 3 shows the complete grammar of base types and shape types.

Using the full power of shape types, we can write down the types of macros such as `and`:

$$(\text{expression} \ldots . \, ()) \rightarrow \text{expression}$$

as well as that of core constructs such as `lambda`. The formal parameter list of `lambda` has the shape type

$$(\text{identifier} \ldots . \, (\texttt{case} \, () \; \text{identifier})) \; .$$

3.4 Structural type checking fails

A traditional type checker recursively traverses an abstract syntax tree and synthesizes the type of the tree from its leaves, using algebraic rules. That is, a type checker descends the tree until it reaches a leaf, for which some external agent (e.g., a type environment or a primitive type judgment) specify the type. For each internal node, it then synthesizes the type from the type of the subtrees.

This context-free traversal approach does not work for shape checking macros and macro uses. Consider the following excerpt from a Scheme program:

```
(add1 x)
```

Since `add1` and `x` are identifiers, one could easily mistake this S-expression for an expression. Suppose, however, that the S-expression appears as the formals part of a `lambda`:

```
(lambda (add1 x) y)
```

Based on this context, we really need to understand the original S-expression as a list of identifiers.

One idea for fixing this problem is traverse the tree and to identify each macro application. Then, it seems possible to check each macro application independently. Put differently, such a type checker would treat each macro application as atomic within the surrounding context and would use traditional type checking locally. Unfortunately, this approach doesn't work either, because macros may dissect their arguments in unforeseen ways. Take a look at the expression

```
(amacro (bmacro x))
```

Assume that `amacro` and `bmacro` are the markers for defined macros of one subexpression each. That is, the S-expression seems to consist of two macro applications. Hence the revised type checker would analyze `(bmacro x)` as a macro application, determining that its type is, say, expression. But take a look at the definition of `amacro`:

```
(define-syntax amacro
  (syntax-laws expression
    [(_ (anything a))
     ((anything any) (a identifier))
     (lambda (a) (quote anything))]))
```

The `amacro` "destroys" the `bmacro` application and instead uses the parts in unexpected ways. More generally, the type checker has thrown away too much information. To determine what to do with the syntax inside of the `amacro`-application, we must use information about `amacro`. Type checking must proceed in a context-sensitive manner.

For a final problem with the conventional type-checking approach, let us examine the `syntax-laws` definition of `let` with unspecified types for the construct's body:

```
(define-syntax let
  (syntax-laws expression
    [(_ ((lhs rhs) ...) body0 body ...)
     ((lhs identifier)
      (rhs expression)
      (body0 ???)
      (body ???))
     ((lambda (lhs ...) body0 body ...) rhs ...)]))
```

Standard Scheme requires that `let`'s body consists of an arbitrarily long sequence of definitions and expressions, but at least one expression. Using the conventional pattern for `let`, we cannot express this constraint. The pattern `body0 body ...` says only that the sequence has to have at least a first element. We can overcome this problem in our model by using the full power of shape types:

```
(define-syntax let
```

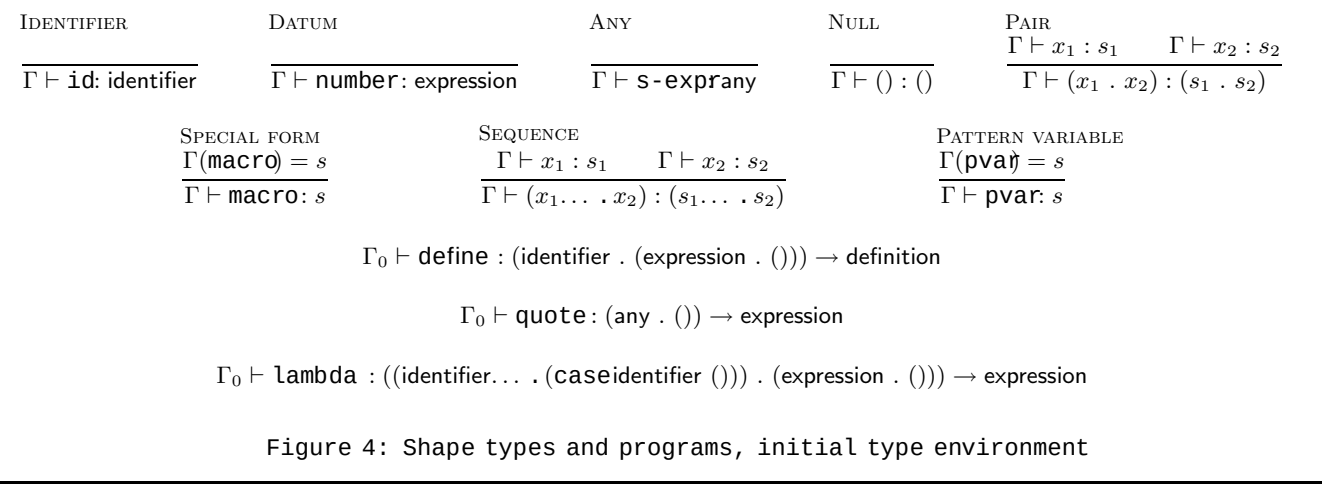


Figure 4: Shape types and programs, initial type environment

```
(syntax-laws ()
  [(_ ((lhs rhs) ...) . body)
   ((lhs identifier)
    (rhs expression)
    (body (definition ...
           expression ...
           expression)))
   ((lambda (lhs ...) body0 body ...) rhs ...)]))
```

This shows that type checking not only must proceed in an unusual context-sensitive manner but that it must also take into account general shapes.

3.5 Relating syntax and shape types

Type checking macros is a matter of verifying that the argument S-expression is below the shape type that describes the domain of the macro. To this end we must specify how patterns, templates, and ordinary top-level expressions give rise to shape types and how types relate to each other.

A macro's domain type is determined by the shape type of the patterns and the shape types of the pattern variables. Specifically, the type of a pattern is the shape type that results from replacing the pattern variables in the pattern with their (guard) types. The type of the entire macro is constructed from the set of patterns' shape types and the result type annotation. Figure 5 formalizes this relationship.

To type-check a top-level expression, our type checker constructs a shape type that describes the structure of the fragment. A pair in the fragment is represented by a pair type, a null by the null type, identifiers by identifier, bound macro keywords by their corresponding arrow types, numbers by expression, and everything else as any. Primitive special forms are treated exactly the same as macros. An initial type environment holds maps every special form to an arrow type.

To type-check a template, the type checker proceeds as for regular top-level expressions, except that it needs to deal with two complications. First, it needs to include pattern variables, which have the types specified in the guards for that clause. Second, it must cope with ellipses, which may appear in various forms and with fewer restrictions than in the pattern.

Figure 4 gives the rules for constructing types for regular program fragments and templates.

4. SHAPE CHECKING

Translating the ideas of the previous section into a working algorithm requires three steps. In this section, we describe these steps, that is, how to check a complete program, what to consider for the subtyping check, and how to implement the check.

4.1 Checking programs

A program shape-checks if its macro definition templates and program body respect the types of its macro applications. The checking of the entire program proceeds in three stages.

First, the type checker builds a type environment from the macro definitions. The type environment maps macro keywords to arrow types. The shape type of a macro is determined by its return type, its patterns, and its guards. The resulting type environment extends the initial type environment with the bindings created via $\vdash_{\mathcal{M}}$.

Second, the type checker verifies that each macro template produces the promised kind of result, assuming it is applied to the specified shapes. For the verification of a template, the global type environment is augmented with the guards in the containing clause.

Third, the type checker verifies that each top-level form in the program is well-shaped. Since a top-level form can be either an expression or a definition, the top-level form is checked against the shape type (case expression definition).

The first step has been described earlier. The third is a simple version of the second due to the macro's guards and the template's ellipses. Hence, the type checker turns the template or top-level form into its corresponding shape type and then determines whether the derived shape is a subtype of the expected shape.

$$\frac{\Gamma(\mathsf{pvar}) = s}{\Gamma \vdash_{\mathcal{P}} \mathsf{pvar} : s} \qquad \frac{}{\Gamma \vdash_{\mathcal{P}} () : ()} \qquad \frac{\Gamma \vdash_{\mathcal{P}} x_1 : s_1 \qquad \Gamma \vdash_{\mathcal{P}} x_2 : s_2}{\Gamma \vdash_{\mathcal{P}} (x_1 \ . \ x_2) : (s_1 \ . \ s_2)} \qquad \frac{\Gamma \vdash_{\mathcal{P}} x : s}{\Gamma \vdash_{\mathcal{P}} (x \ldots \ . \ ()) : (s \ldots \ . \ ())}$$

$$\frac{G_i \vdash_{\mathcal{P}} P_i : s_i \qquad i \leq n}{\vdash_{\mathcal{M}} (\mathsf{define\text{-}syntax}\ m\ (\mathsf{syntax\text{-}laws}\ t\ ((_ \ . \ P_0)\ G_0\ T_0) \cdots (_ \ . \ P_n)\ G_n\ T_n)) \blacktriangleright (\mathsf{case}\ s_0 \cdots\ s_n) \to t}$$

Figure 5: Shape types and macro definitions

Regular application
$(\mathsf{expression} \ . \ (\mathsf{expression} \ldots \ . \ ())) \leq \mathsf{expression}$

Special form application
$$\frac{s' \leq s}{(s \to t \ . \ s') \leq t}$$

Figure 6: Shape type simplification

The subtype relation for shape types is the natural generalization of the subtype relation on base types to the shape types plus two additional subtyping rules (see figure 6).

4.2 Subtyping

Our algorithm generalizes Amadio and Cardelli's recursive subtyping algorithm using cyclicity tests [1]. It is not a plain structural recursion on the two types. Two issues complicate the algorithm. One arises from the way pair shapes and case shapes interact, and the other from sequence shapes.

It is useful to think of the right hand side of the comparison not as a single type, but as a set of possible choices. The set increases as different possibilities are introduced by case and sequence types. It is not sufficient to check whether the type on the left matches any one type in the set. It may be that the type on the left may be covered only by the combination of multiple types on the right.

The following two inequalities illustrate how `case` and `pair` types interact (a, b, and c are incomparable types):

$$\begin{aligned} (a \ . \ (\mathsf{case}\, b\ c)) &\leq (\mathsf{case}(a \ . \ b)\ (a \ . \ c)) \\ ((\mathsf{case}\, a\ b) \ . \ c) &\leq (\mathsf{case}(a \ . \ c)\ (b \ . \ c)) \end{aligned}$$

In the first case, we need to check both the `car` and the `cdr` of the pair on the left. The question is to which type on the right we need to compare them. Clearly, this inequality test should succeed, but if we divide the set and consider the `cdr` of each pair separately, the algorithm fails to verify the inequality, because $(\mathsf{case}\, b\ c) \not\leq b$ and $(\mathsf{case}\, b\ c) \not\leq c$. The second case is the dual of the first and shows that we cannot split the set when we test the `car` of a pair.

One solution seems to be to check the `car` of the left with the `car`s of all the pairs on the right, and to check the `cdr` of the left with the `cdr`s of all the pairs on the right. Unfortunately, that solution is unsound. It accepts the following bad inequality

$$(a \ . \ a) \leq (\mathsf{case}(a \ . \ b)\ (b \ . \ a)) \ ,$$

because the `car` of the first option would match and the `cdr` of the second.

The correct solution is to match not a set of types on the right, but a set of states, where a state is either $\bullet$ (the initial matching context) or a type with a state as context. The state's context describes the state that is made available to the set of `cdr`s to match if the state's type is matched. The context is only extended when checking pairs. The example above becomes:

$$(a \ . \ a) \leq (\mathsf{case}(a \ . \ b)^{\bullet}\ (b \ . \ a)^{\bullet})$$

Checking the `car` of the pair becomes

$$a \leq (\mathsf{case} a^{b,\bullet}\ b^{a,\bullet})$$

The first state in the set matches and the second fails. So the `cdr` is matched:

$$a \leq b^{\bullet}$$

which fails as required.

The second complication arises because of sequences. When a sequence $(s_r \ldots \ . \ s_u)$ is encountered on either the left or the right, it is unfolded into $(\mathsf{case}\, s_u\ (s_r \ . \ (s_r \ldots \ . \ s_u)))$. The algorithm relies on a trace accumulator to detect cycles in checking. The trace keeps track of what inequality checks are currently under consideration. For example, the call to check $(a \ldots \ . \ b) \leq (a \ldots \ . \ b)$ would unfold both sequences, check their base cases, check their `car`s, and then return to checking the same inequality. Since that combination of type and set of states is in the trace accumulator, a cycle has occurred and it is correct to succeed [1].

4.3 Shape checking at work

Recall the macro `++` from Section 2. The following is the macro written in our language:

```
(define-syntax ++
  (syntax-laws expression
    [(_ x)
     ([x identifier])
     (begin (set! x (add1 x)) x)]))
```

This gives `++` the following shape type: [3]

$$(\mathsf{identifier} \ . \ ()) \to \mathsf{expression}$$

In this context, the macro application

```
(++ (vector-ref v 0))
```

[3] We abbreviate $(\mathsf{case} s)$ as s.

is invalid, because the shape checking algorithm cannot show that the shapes of ++'s input are below its input shape.

The most specific shape of the input is

((identifier . (identifier . (expression . ()))) . ())

This shape is not bbelow (identifier . ()), the input shape of the macro. Thus the macro application of ++ is flagged as erroneous instead of the `set!` special form application that it expands into.

Checking is also performed on the templates of a macro definition. In the following definition of `where`, `lhs` and `rhs` are mistakenly swapped in the template:

```
(define-syntax where
  (syntax-laws expression
    [(_ body lhs is rhs)
     ([body expression]
      [lhs identifier]
      [rhs expression])
     (let ([rhs lhs]) body)]))
```

In order to match the shape of the template with expression, the shape checker needs to prove expression $\leq$ identifier, to satisfy the input shape of `let`. Since this inequality is not true, shape checking fails for the template. The macro definition is therefore rejected, even without any uses of the `where` macro.

```
;; A TypeSymbol is one of
;; 'identifier, 'expression, 'definition, 'any

;; A Type is
;;   - (make-base-type TypeSymbol)
;;   - (make-null-type)
;;   - (make-pair-type Type Type)
;;   - (make-sequence-type Type Type)
;;   - (make-arrow-type Type Type)
;;   - (make-case-type [Type])
(define-struct base-type (symbol))
(define-struct null-type ())
(define-struct pair-type (car cdr))
(define-struct sequence-type (rep final))
(define-struct arrow-type (domain range))
(define-struct case-type (cases))

;; A State is
;;   - (make-done-state)
;;   - (make-state Type State)
(define-struct done-state ())
(define-struct state (type context))
```

Figure 7: Data definitions

4.4 Implementation

This section presents the algorithm that determines whether one shape type is a subtype of another. We start with the data definitions and follow with the interface procedures. The last part covers those procedures that perform the recursive traversals.

Figure 7 shows the data definitions. We represent shape types as structures and base types as symbols wrapped in a `base-type` structure.

The main function is the procedure `subshape?`, which constructs a state from the type on the right hand side of the inequality to be tested and calls `check` to conduct the actual comparison.

The subshape checking algorithm maintains the invariant that the set of states representing the right hand side never contains a type whose outermost type constructor is sequence or case. Sequences are unfolded to their final type and a pair of their repeated type and the sequence type. The variants of a case type are absorbed into the set of states. The procedure `normalize` is responsible for maintaining this invariant.

The `check` procedure consumes a trace, a type, and a list of states. It produces a list of states to be used to check the `cdr` of a pair, as described above. If `check` produces a list containing the `done` state, checking has succeeded. Otherwise, it returns the empty list to indicate failure.

The `check` procedure always first consults the trace to detect and escape from cycles. If no cycle is found, `check` calls to `check/shape` with all the shape types and `check/base` with only the base types. The union of the results is returned.

The `unionmatch` macro checks the value of an expression against the pattern of each clause. For each pattern which succeeds, it evaluates the body and returns the union of the results.

The procedure `check/shape` performs a straightforward case analysis on the composite shape types. A null type matches exactly null types. A pair type matches the `car` against the `car`s of all pairs in the state set. The function `abstract-car` takes the `car` of all pairs in the set and extends the matching context for each with that pair's `cdr`. A sequence type matches if both cases of its unfolded representation match. A case type matches if all of its variants match. Since we must return a set of states, we use `union` and `intersection` rather than simple `or` and `and`.

The procedure `check/base` verifies subtyping for basic types. Any type is under `any`. Two base types are compared using the simple subtype relation for base types. An expression can be formed by a pair of an expression and a sequence of expressions, and either expression or definition can be the result of the appropriate special form application.

The procedure `check/macro` checks a macro application, using the shape types of the macro keyword; `normalize` maintains the invariant stated above; and `abstract-car` takes the `car` of all pair types and extends the context of the resulting states.

```
;; subshape? : Type Type -> boolean
(define (subshape? lhs rhs)
  (ormap done-state?
    (check empty-trace lhs
           (normalize (make-state rhs (make-done-state))))))

;; check : Trace Type [State] -> [State]
(define (check trace lhs states)
  (or (trace-lookup trace lhs states)
      (let [(trace (extend-trace trace lhs states))
            (base-states (filter state/base-type? states))]
        (union*
         (cons
          (check/shape trace lhs states)
          (map (lambda (bstate) (check/base trace lhs bstate))
               bstates))))))
```

Figure 8: Driving procedures

```
;; check/shape : Trace Type [State] -> [State]
(define (check/shape trace lhs states)
  (unionmatch lhs
    [($ null-type)
     (union* (map (lambda (s) (if (null-type? (state-type s)) (succeed s) (fail))) states))]
    [($ pair-type lhs-car lhs-cdr)
     (let [(cdrstates (check trace lhs-car (abstract-car states)))]
       (check trace lhs-cdr cdrstates))]
    [($ sequence-type lhs-rep lhs-final)
     (intersect (check trace lhs-final states)
                (check trace (pair lhs-rep lhs) states))]
    [($ case-type lhs-cases)
     (intersect (map (lambda (lhs-case) (check trace lhs-case states)) lhs-cases))]))

;; check/base : Trace Type State -> [State]
(define (check/shape trace lhs base-state)
  (unionmatch (state-type base-state)
    [($ base-type 'any) (succeed base-state)]
    [($ base-type b)
     (if (and (base-type? lhs) (subtype? lhs (base-type b)))
         (succeed base-state)
         (fail))]
    [($ base-type 'expression)
     (check trace lhs
            (list (make-state
                   (pair-type expression (sequence-type expression (null-type)))
                   (state-context base-state))))]
    [($ base-type (or 'expression 'definition))
     (if (and (pair-type? lhs) (arrow-type? (pair-type-car lhs)))
         (check/macro trace lhs base-state)
         (fail))]))
```

Figure 9: check/shape and check/base

```
;; check/macro : Trace Type State -> [State]
(define (check/macro trace lhs base-state)
  (let [(macro (pair-type-car lhs))
        (argument (pair-type-cdr lhs))]
    (if (base-type-equal? (arrow-type-range macro) (state-type base-state))
        (check trace argument
               (make-state (arrow-type-domain macro) (state-context base-state)))
        (fail))))

;; abstract-car : [State] -> [State]
(define (abstract-car states)
  (union (map abstract-car/1 states)))

;; abstract-car/1 : State -> [State]
(define (abstract-car/1 state)
  (let [(type (state-type))]
    (if (pair-type? type)
        (normalize
         (make-state (pair-type-car type)
                     (make-state (pair-type-cdr type) (state-context state))))
        '())))

;; normalize : State -> [State]
(define (normalize state)
  (map (lambda (type) (make-state type (state-context state)))
       (normalize-type (state-type state))))

;; normalize-type : Type -> [Type]
(define (normalize-type type)
  (cond [(sequence-type? type)
         (cons (pair-type (sequence-type-rep type) type)
               (normalize-type (sequence-type-final type)))]
        [(case-type? type)
         (union (map normalize-type (case-type-cases type)))]
        [else (list type)]))
```

Figure 10: check/macro and auxiliary procedures

4.5 Some first experiences

We used an implementation of the algorithm described to check implementations of the special forms described as derived syntax in R^{5}RS [8]. The algorithm was extended to handle literals in patterns.

Defining these forms in `syntax-laws` poses two problems. First, we need to reformulate the definitions to be compatible with our system. Second, we need to write down shape types for each pattern variable.

The macro definitions for the derived syntax given in R^{5}RS were not compatible with our system. For example, the common shape

(definition... . (expression... . (expression . ())))

cannot be expressed with the idiomatic pattern

```
(body0 body ...)
```

used in R^{5}RS; there are no suitable annotations. To solve this problem, we rewrite `let` using a "dotted" pattern:

```
(define-syntax let
  (syntax-laws expression
    [(let ((name val) ...) . body)
     ([name identifier] [val expression]
      [body (definition ...
              expression ... expression)])
     ((lambda (name ...) . body) val ...)]))
```

Shape annotations for simple macros are as easy as type annotations in most languages. Complicated macros such as `cond`, however, have extremely verbose annotations due to the complexity of the syntax of `cond` clauses. We are exploring a method of naming or abbreviating shapes.

5. RELATED WORK, LIMITATIONS, AND FUTURE WORK

Cardelli, Matthes, and Abadi [2] study macros as extensible parsing. They superimpose enough discipline so that parser extensions don't violate the lexical structure of the program. They do not consider the question of whether macro applications are well-shaped but instead ensure that the grammar extension produces a well-defined grammar.

Ganz, Sabry, and Taha [7] present MacroML, a version of ML with an extremely simple form of macros. Their macros require the macro user to specify run-time values, syntax, and binding relationships at the place of macro use. In return, they can type check their macros with a tower of type systems. The type checker verifies that MacroML macros expand properly and that the code they produce type checks in ML. Unfortunately, none of their type-checking techniques for macros carry over to Scheme's macro system because of the simplicity of their assumptions.

We have extended our own work so that it applies to a large portion of Scheme's standard macro system. That is, we can rewrite and type check the macros from the Scheme report in the `syntax-law` notation. The expanded version also covers all core forms with two exceptions: `begin` and `quasiquote`. Still, our system imposes several restrictions on the macro writer, including the need for type declarations and the elimination of macro-arguments.

In the near future, we intend to investigate a soundness theorem for macro expansion similar to type soundness for functional languages. Specifically, macro expansion (in our model) should always produce well-formed syntax modulo context-sensitive constraints (e.g., `(lambda (x x) (+ x 1))`) and ellipsis mismatch. The latter is, of course, is analogous to type checking array lookups; the former is probably beyond the scope of a type discipline.

6. REFERENCES

[1] Amadio, R. and L. Cardelli. Subtyping recursive types. In ACM Transactions on Programming Languages and Systems, volume 15, pages 575–631, 1993.

[2] Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.

[3] Clinger, W. and J. Rees. Macros that work. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 155–162, 1991.

[4] Dybvig, R. K. The Scheme Programming Language. Prentice-Hall, 1 edition, 1987.

[5] 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. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.

[6] Flatt, M. Composable and compilable macros: You want it when? In ACM SIGPLAN International Conference on Functional Programming, 2002.

[7] Ganz, S. E., A. Sabry and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In International Conference on Functional Programming, pages 74–85, 2001.

[8] Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. ACM SIGPLAN Notices, 33(9):26–76, 1998.

[9] Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In ACM Symposium on Lisp and Functional Programming, pages 151–161, 1986.

[10] Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages pages 77–84, 1987.

[11] Queinnec, C. Macroexpansion reflective tower. In Kiczales, G., editor, Proceedings of the R eflection'96 Conference, pages 93–104, San Francisco (California, USA), 1996.

Porting Scheme Programs

Dorai Sitaram
Verizon
40 Sylvan Road
Waltham, MA 02451

Abstract

The Scheme standard and the Scheme reports define not one but an entire family of programming languages. Programmers can still create useful programs in small dialect-specific extensions of the standardized Scheme language but porting such programs from one dialect to another requires tedious work. This paper presents SCMXLATE, a lightweight software tool that automates a large portion of this work.

1. ON THE PORTABILITY OF SCHEME

The existence of the IEEE Scheme Standard [6] appears to suggest that Scheme programmers can write a program once and run it everywhere. Unfortunately, appearances are deceiving. The Scheme standard and the Scheme reports [16, 15, 1, 2, 8] do not define a useful programming language for all platforms. Instead they—like the Algol 60 [9] report—define a family of programming languages that individual implementors can instantiate to a concrete programming language for a specific platform. As a result, Olin Shivers can publicly state that "Scheme is the least portable language I know" without expecting any contradictions from the authors of the standard or report documents.

Even though the Scheme standard and reports define a minimal language, it is still possible to write useful programs in small extensions of the standard language.[1] To understand the expressive power of standard Scheme plus a small library, take a look at SLaTEX [10], a package for rendering Scheme code in an Algol-like presentation style via TEX (approximately 2,600 lines of code), and TEX2page [11], a package for rendering TeX documents as HTML (approximately 9,200 lines of code).

To create a stand-alone application from a Scheme program in some different dialect of Scheme, programmers must often conduct a systematic three-stage transformation. First, they need installation-specific configuration code. Second, they add code for functions that the targeted dialect doesn't support. Finally, they must perform a number of tedious and labor-intensive surgery on the code itself.

This short paper presents SCMXLATE,[2] a program that assists programmers with the task of porting programs from one Scheme dialect to another. Specifically, the program assembles new packages from existing packages, libraries and directives. The program has been applied to a number of packages, including the above-mentioned SLaTEX and TEX2page.

The next section presents the general model of porting code. The third section describes the "surgery" directives that have proven useful for porting a number of large packages among several Scheme dialects. The last two sections discuss related and suggestions for future work.

2. PORTING PROGRAMS

SCMXLATE provides two services. First, it assists programmers with the tedium of porting Scheme programs from one dialect to another. Second, it provides the ability to configure a program into an installation-specific application.

In the first subsection, we present SCMXLATE's underlying assumptions about programs and the conversion process. In the second subsection, we illustrate an end-user's experience with SCMXLATE-based packages. In the third subsection, we describe how SCMXLATE translates a program from one Scheme dialect to another and how it assists with the creation of a full-fledged application.

2.1 Assumptions

Standard Scheme does not provide a module mechanism for partitioning a program into several components with well-specified dependencies. Instead, the Scheme standard implies that programmers treat files as components and combine them using Scheme's `load` instruction.

Since Scheme does not specify a method for describing directory paths in a platform-independent manner, SCMXLATE assumes that the programmer has placed all files into a single directory. Figure 1 displays an example. The sample program consists of three files, which are displayed in *italic*

[1] We use "Standard Scheme" for both the IEEE language and the language defined in the reports.

[2] We suggest reading the name as SCM × LATE and pronouncing it as "skim latte".

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Dorai Sitaram.

pgdir/	the package directory
apple	a source file
banana.ss	...
orange.scm	...
`dialects/`	the port directory
`files-to-be-ported.scm`	specifies the files in the parent directory that must be converted
`dialects-supported.scm`	specifies the target dialects
GUILE-APPLE	specifies dialect-specific instructions for a to-be-converted source file
GUILE-BANANA.SS	...
GUILE-ORANGE.SCM	...
SCMXLATE-APPLE	specifies installation-specific adjustments for a to-be-converted source file
SCMXLATE-BANANA.SS	...
SCMXLATE-ORANGE.SCM	...
my-apple	a generated target file
my-banana.ss	...
my-orange.scm	...

Figure 1: A sample directory organization

font. [Note to readers: please ignore the rest of the figure for now.]

A program is not an application. To create an application from a program, the installer must often specify some values that depend on the context in which the program runs. For example, a spelling program may need to know about some idiosyncratic words for a specific user. While an interactive approach works well for a spelling program, it is a terrible idea for a Unix-style filter, which transforms a text file in one format into another one. For such programs, it is best if users conduct a configuration process that creates the installation-specific defaults. SCMXLATE assumes that this configuration step should be a part of the installation and port process and therefore supports it in a minimal manner, too.

2.2 Installing a Package with SCMXLATE

Assume that a programmer has prepared some package for use on several Scheme dialects and possibly different platforms. Also imagine an installer who wishes to install the package for a Scheme dialect that is different from the source language and for a new platform. This installer must take two steps.

First, the user must install SCMXLATE on the target platform. Second, the user must configure the actual package. To do so, the user launches the target Scheme implementation in the package directory and types

```
(load "/usr/local/lib/scmxlate/scmxlate.scm")
```

where the `load` argument uses the full pathname for the directory that contains SCMXLATE. As it is loaded, SCMXLATE poses a few questions with a choice of possible answers, including a question that determines the target dialect,[3] though a knowledgeable user can provide different answers.

[3]The Scheme standard and reports do not provide a generic mechanism for Scheme programs to determine in which dialect they run.

When all the questions are answered, SCMXLATE creates the platform-specific and dialect-specific package. Naturally, the programmer can also prepare versions of a package for various dialects directly.

2.3 Preparing a Package for SCMXLATE

A programmer who wishes to distribute a package for use with different Scheme dialects creates a sub-directory with several files in the package directory. The files specify the pieces of the package that require translation, the dialects that are supported, and optional dialect-specific preambles for each file that is to be translated.

If the package also requires installation-specific configuration instructions, the programmer supplies files in the package directory. Specifically, the programmer creates one file per source file that requires special configurations. These additional files are independent of the target dialect but may contain SCMXLATE rewriting directives that process the corresponding source file (see the next section).

Let us refine our example from figure 1. Assume the source language is MzScheme and the file *apple* uses the library function

```
file-or-directory-modify-seconds
```

Also assume that the target language is Guile. Then the dialect-specific transformation file for *apple*—GUILE-APPLE in the figure—should contain the following Guile definition:

```
(define file-or-directory-modify-seconds
  (lambda (f)
    (vector-ref (stat f) 9)))
```

If the dialect-configuration file supplies a definition for a name that is also defined in the input file, then the output file contains the definition from the dialect-configuration file, not the input file. For example, suppose *apple* contains the

MzScheme definition

```
(define file-newer?
  (lambda (f1 f2)
    ;checks if f1 is newer than f2
    (> (file-or-directory-modify-seconds f1)
       (file-or-directory-modify-seconds f2))))
```

In Guile, this definition is expressed as

```
(define file-newer?
  (lambda (f1 f2)
    ;checks if f1 is newer than f2
    (> (vector-ref (stat f1) 9)
       (vector-ref (stat f2) 9))))
```

and this definition is therefore placed into GUILE-APPLE. Then SCMXLATE's translation of *apple* directly incorporates the Guile definition into the output file. That is, SCMXLATE doesn't even attempt to translate the MzScheme definition of the same name in the input file.

Let us revisit figure 1. In addition to the source files, the figure displays the complete directory structure for a specific example. SCMXLATE inspects the file and directory names in `type-writer` font for instructions on how to translate the source files in *pgdir*. In particular,

`files-to-be-ported.scm` contains strings that specify the names for those files that SCMXLATE must translate;

`dialects-supported.scm` contains symbols, which specify the names of the dialects for which the programmer has prepared translations; currently, SCMXLATE supports

BIGLOO,	PETITE,
CHEZ,	PSCHEME,
CL,	SCHEME48,
GAMBIT,	SCM,
GAUCHE,	SXM,
GUILE,	SCSH,
KAWA,	STK,
MITSCHEME,	STKLOS, and
MZSCHEME,	UMBSCHEME.

To provide file-specific adaptation code per dialect, the programmer creates a file name with a dialect-indicating prefix; in figure 1 these files are displayed in SMALL-CAPS font. Finally, installation-specific configuration code is in files whose names are prefixed with SCMXLATE-. All SMALL-CAP files are optional.

When SCMXLATE is run in the *pgdir* directory, it creates one file per source file. In figure 1, these files appear in *underlined italic* font. Figure 2 shows the structure of these generated files. The installation-specific code appears at the very top of the file; it is followed by the dialect-specific code. The bottom part of the file consists of the translated source code. The translation process is specified via directives that comes with the installation-specific and dialect-specific pieces.

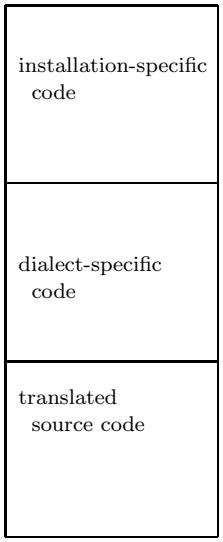


Figure 2: The file structure

3. THE DIRECTIVES

In addition to Scheme code intended to either augment or override code in the input file, the dialect-configuration and installation-configuration files can use a small set of directives to finely control the text that goes into the output file, and even specify actions that go beyond the mere creation of the output file. These directives are now described.

3.1 `scmxlate-insert`

As we saw, Scheme code in the dialect-configuration and installation-configuration files is transferred verbatim to the output file. Sometimes, we need to put into the output file arbitrary text that is not Scheme code. For instance, we may want the output file to start with a "shell magic" line, so that it can be used as a shell script. Such text can be written using the `scmxlate-insert` directive, which evaluates its subforms in Scheme and displays them on the output file.

Thus, if the following directive occurs at the top of GUILE-APPLE

```
(scmxlate-insert
  "#!/bin/sh
exec guile -s $0 \"$@\"
!#
")
```

the output file `my-apple` for the Guile-specific version of the package starts with the line

```
#!/bin/sh
exec guile -s $0 "$@"
!#
```

Note that the order of the code and `scmxlate-insert` text in the configuration file is preserved in the output file.

3.2 scmxlate-postamble

Typically, the original Scheme code (augmented with the code of scmxlate-inserts) occurs in the output file before the translated counterpart of input file's contents, and thus may be considered as *preamble* text. Sometimes we need to add *postamble* text, ie, things that go *after* the code from the input file. In order to do this, place the directive

```
(scmxlate-postamble)
```

after any preamble text in the dialect-configuration file. Everything following that line, whether ordinary Scheme code or scmxlate-inserts, shows up in the output file after the translated contents of the input file.

3.3 scmxlate-postprocess

One can also specify actions that need to performed after the output file has been written. Say we want the Guile output file for *apple* to be named *pear* rather than *my-apple*. We can enclose Scheme code for achieving this inside the SCMXLATE directive scmxlate-postprocess:

```
(scmxlate-postprocess
  (rename-file "my-apple" "pear"))
```

3.4 scmxlate-ignore-define

Sometimes the input file has a definition that the target dialect does not need, either because the target dialect already has it as a primitive, or because we wish to completely rewrite input code that uses that definition. That is, if the target dialect is MzScheme, which already contains reverse!, any definition of reverse! in the input file can be ignored.

```
(scmxlate-ignore-define reverse!)
```

The scmxlate-ignore-define form consumes any number of names, and all corresponding definitions are ignored.

3.5 scmxlate-rename

Sometimes we want to rename certain identifiers from the input file. One possible motivation is that these identifiers name nonstandard primitives that are provided under a different name in the target dialect. For instance, the MzScheme functions

```
current-directory ; -> String
file-or-directory-modify-seconds ; String -> Number
```

are equivalent to the Bigloo functions

```
chdir ; -> String
file-modification-time ; String -> Number
```

respectively. So if the MzScheme input file uses these functions, the Bigloo dialect-configuration file should contain

```
(scmxlate-rename
 (current-directory
  chdir)
 (file-or-directory-modify-seconds
  file-modification-time))
```

Note the syntax: scmxlate-rename has any number of twosomes as arguments. The left item is the name in the input file, and the right item is its proposed replacement.

3.6 scmxlate-rename-define

Sometimes the input file includes a definition for an operator that the target dialect already has as a primitive, but with a different name. That is, consider an input file that contains a definition for nreverse. MzScheme has the same operator but with name reverse!, which means that the MzScheme dialect-configuration file should contain the following directive:

```
(scmxlate-rename-define
  (nreverse reverse!))
```

Note that this is shorthand for

```
(scmxlate-ignore-define nreverse)
(scmxlate-rename
  (nreverse reverse!))
```

3.7 scmxlate-prefix

Another motivation for scmxlate-rename is to avoid polluting namespace. We may wish to have short names in the input file, but when we configure it, we want longer, "qualified" names. It is possible to use scmxlate-rename for this action, but the scmxlate-prefix is convenient when the newer names are all uniformly formed by adding a prefix.

Thus,

```
(scmxlate-prefix
  "regexp::"
  match
  substitute
  substitute-all)
```

renames

match to regexp::match,

substitute to regexp::substitute,

and

substitute-all to regexp::substitute-all, respectively.

The first argument of scmxlate-prefix is the string form of the prefix; the remaining arguments are the identifiers that should be renamed.

3.8 scmxlate-cond

Sometimes we want parts of the dialect-configuration file to be processed only when some condition holds. For instance, we can use the following cond-like conditional in a dialect-configuration file for MzScheme to write out a shell-magic line appropriate to the operating system:

```
(scmxlate-cond
  ((eqv? (system-type) 'unix)
   (scmxlate-insert *unix-shell-magic-line*))
  ((eqv? (system-type) 'windows)
   (scmxlate-insert *windows-shell-magic-line*)))
```

In this expression, the identifiers `*unix-shell-magic-line*` and `*windows-shell-magic-line*` must denote appropriate strings.

Note that while `scmxlate-cond` allows the `else` keyword for its final clause, it does not support the `=>` keyword of standard Scheme's `cond`.

3.9 `scmxlate-eval`

The test argument of `scmxlate-cond` and all the arguments of `scmxlate-insert` are evaluated in the Scheme global environment when SCMXLATE is running. This environment can be enhanced via `scmxlate-eval`. Thus, if we had

```
(scmxlate-eval
  (define *unix-shell-magic-line* <...>)
  (define *windows-shell-magic-line* <...>))
```

where the `<...>` stand for code that constructs appropriate strings, then we could use the two variables as arguments to `scmxlate-insert` in the above example for `scmxlate-cond`.

A `scmxlate-eval` expression can have any number of subexpressions. It evaluates all of them in the given order.

3.10 `scmxlate-compile`

`scmxlate-compile` can be used to tell if the output file is to be compiled. Typical usage is

```
(scmxlate-compile #t)      ;or
(scmxlate-compile #f)
```

The first forces compilation but only if the dialect supports it, and the second disables compilation even if the dialect supports it. The argument of `scmxlate-compile` can be any expression, which is evaluated only for its boolean significance.

Without a `scmxlate-compile` setting, SCMXLATE asks the user explicitly for advice, but only if the dialect supports compilation.

3.11 `scmxlate-include`

It is often convenient to keep some of the text that should go into a dialect-configuration file in a separate file. Some definitions may naturally be already written down somewhere else, or we may want the text to be shared across several dialect-configuration files (for different dialects). The call

```
(scmxlate-include "filename")
```

inserts the content of `"filename"` into the file.

3.12 `scmxlate-uncall`

It is sometimes necessary to skip a top-level call when translating an input file. For instance, the input file may be used as a script file whose scriptural action consists in calling a procedure called `main`. The target dialect may not allow the output file to be a script, so the user may prefer to load the output file into Scheme as a library and make other arrangements to invoke its functionality. To disable the call to `main` in the output file, add

```
(scmxlate-uncall main)
```

to the configuration file.

The `scmxlate-uncall` form consumes any number of symbol arguments. All top-level calls to these functions are disabled in the output.

4. RELATED WORK

SCMXLATE wouldn't be necessary if standard Scheme were a practical language. One way to achieve practicality is to equip a language with powerful, expressive libraries and extensions. Jaffer's SLIB [7] effort and the SRFI process [13] aim to supplement Scheme in just such a way. If they are successful, the various Scheme dialects will resemble each other as far as the source language itself is concerned, thus rendering a good part of SCMXLATE obsolete.

From a reasonably abstract perspective, SCMXLATE provides those services to Scheme that `autoconf` [5] provides to C. Both use preprocessing to conduct tests on the code, to assist the target compiler, and to create proper contexts for the ported program. Naturally, `autoconf` is a more expressive and more encompassing tool than SCMXLATE; it has been around for twice as long.

5. SUMMARY

The paper explains how SCMXLATE assists programmers with the translation of Scheme programs from one dialect to another. The software tool evolved due to the demand to translate various packages into a number of different dialects. It is now easy to use and robust. Indeed, SCMXLATE can now also translate Scheme programs into Common Lisp programs [14] though the resulting code is somewhat unnatural.

Although SCMXLATE has become an accessible product of its own, it still lacks good environmental support. A programmer preparing a package for SCMXLATE could make good use of sophisticated syntax coloring tools such as those provided in DrScheme [3] and refactoring tools such as Dr. Jones [4], especially if they are integrated with the programming environment.

SCMXLATE is available on the Web. For more information, the interested reader should consult the on-line manual [12].

6. ACKNOWLEDGMENT

I thank Matthias Felleisen for helpful discussions and for shaping the goals of SCMXLATE.

7. REFERENCES

[1] Clinger, W. The revised revised report on the algorithmic language Scheme. Joint technical report, Indiana University and MIT, 1985.

[2] Clinger, W. and J. Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[3] 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, 2002.

[4] Foltz, M. A. Dr. Jones: A software design explorer's crystal ball. Technical report, MIT AI Lab, expected in 2003.

[5] GNU. Autoconf, `http://www.gnu.org/software/autoconf/` 1998–2003.

[6] Haynes, C. Standard for the Scheme programming language. *IEEE Document P1178/D5*, October 1991.

[7] Jaffer, A. SLIB, `http://www.swiss.ai.mit.edu/~jaffer/SLIB.html` 1995–2003.

[8] Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[9] Naur, P. E. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[10] Sitaram, D. How to use SLaTEX, `http://www.ccs.neu.edu/home/dorai/slatex/slatxdoc.html` 1990.

[11] Sitaram, D. TEX2page, `http://www.ccs.neu.edu/home/dorai/tex2page/tex2page-doc.html` 2000.

[12] Sitaram, D. scmxlate, `http://www.ccs.neu.edu/home/dorai/scmxlate/scmxlate.html` 2000.

[13] Sperber, M., D. Rush, F. Solsona, S. Krishnamurthi and D. Mason. Scheme requests for implementation, `http://srfi.schemers.org/` 1998–2003.

[14] Steele Jr., G. L. *Common Lisp—The Language*. Digital Press, 1984.

[15] Steele Jr., G. L. and G. L. Sussman. The revised report on scheme, a dialect of lisp. Technical Report 452, MIT Artificial Intelligence Laboratory, 1978.

[16] Sussman, G. L. and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.