MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1427 October, 1993

# Translucent Procedures, Abstraction without Opacity

**Guillermo J. Rozas**



This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097, and by the National Science Foundation under grant number MIP-9001651.

# Translucent Procedures, Abstraction without Opacity

by

Guillermo J. Rozas

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 1993, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

This report introduces *translucent procedures* as a new mechanism for implementing behavioral abstractions. Like an ordinary procedure, a translucent procedure can be invoked, and thus provides an obvious way to capture a *behavior*. Translucent procedures, like ordinary procedures, can be manipulated as first-class objects and combined using functional composition. But unlike ordinary procedures, translucent procedures have structure that can be examined in well-specified non-destructive ways, without invoking the procedure.

I have developed an experimental implementation of a normal-order lambda-calculus evaluator augmented with novel reflection mechanisms for controlled violation of the opacity of procedures. I demonstrate the utility of translucent procedures by using this evaluator to develop large application examples from the domains of graphics, computer algebra, compiler design, and numerical analysis.

Thesis Supervisor: Harold Abelson
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Gerald J. Sussman
Title: Matsushita Professor of Electrical Engineering

*A* **Guillermo Rozas Mazo** *(1931–1990),*

*cuyo deseo de dar a sus hijos la educación que él no pudo recibir,*

*con este trabajo realizo.*

*Lo prometido es deuda.*

# Acknowledgments

There are many people to whom I owe my sanity and to whom I am indebted. In no particular order, I would like to acknowledge:

My mother, Juana Rodríguez Artés, and my sister, Mª Librada Rozas Rodríguez, who have put up with me longer than anyone else, and, somehow, they are not tired of me yet.

Barb and Jim Miller, for encouraging me, for mothering me, for trying out and making Spanish food, and for showing me that there is life beyond MIT and Tech. Sq.

Gerry Sussman and Hal Abelson, for being friends, for being tolerant, for financially supporting me, and for creating an environment where people can have fun together even if they work on completely unrelated subjects.

*The lunch crowd*: Arthur Gleckler, Brian LaMacchia, and Mark Friedman. Switzerland revolves around you.

Rocky Cardalisco, who periodically pulls me away from MIT to prove to me that the outside world still exists.

Kleanthes Koniaris, for debugging my writing.

Franklyn Turbak, for debugging my thinking.

Chris Hanson, for MIT Scheme and for Edwin.

Henry Wu, for porting MIT Scheme to MS-DOS, enlightening me about PCs, and encouraging me to buy one, without which I would still be getting nasty letters from the EECS Area II committee.

Andy Berlin, for pushing me along by threatening to graduate before me.

Michael Blair, k.p.a. Ziggy, and Stephen Adams, for liking beer, and for reminding me that I do too.

Rajeev Surati, for sharing his office with a maniac fighting with LaTeX.

Natalya Cohen, Jason Wilson, and the other Switzerland undergraduates. Not only do you boost graduate students' egos by being easily impressed, but you remind us how much fun it was to learn when everything was new.

The rest of Switzerland.

The MIT Lecture Series Committee, LSC, for being an immensely fun activity, for giving me something to do unrelated to my work, and for teaching me what it's like to have one thousand people ready to call you all sorts of names when you make a tiny mistake.

Last but not least, my relatives, especially my grandfather, Juan Rodríguez García, who has continually asked me when I was going to grow up and graduate.

# Contents


**1 Abstraction and Procedural Opacity** **1**

1.1 Henderson's Picture Language . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Limitations of Procedural Representations . . . . . . . . . . . . . . . 7

1.3 Overcoming Opacity . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

1.4 The Picture Language with Translucent Procedures . . . . . . . . . . 12

1.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**2 Translucent Procedures and the Procedural Pattern Matcher** **21**

2.1 `tlambda` and TScheme . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.1.1 Fundamental Differences . . . . . . . . . . . . . . . . . . . . . . 23

2.1.2 Accidental Differences . . . . . . . . . . . . . . . . . . . . . . . 26

2.2 The Procedural Pattern Matcher . . . . . . . . . . . . . . . . . . . . 31

**3 Solvers for Systems of Equations** **41**

3.1 Solving Systems by Substitution . . . . . . . . . . . . . . . . . . . . 41

3.2 Substitution as an Operation on Functions . . . . . . . . . . . . . . . 43

3.3 Additional Concerns to the Application of the Method . . . . . . . . 46

3.4 The Solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

3.5 The Limitations of Opaque Procedures . . . . . . . . . . . . . . . . . 55

3.6 Overcoming Opacity . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

3.7 Loose Ends . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

3.8 Summary . . . . . . . . . . 63

**4 Metacircular Interpreters and Compilers** **65**

4.1 Why Use Interpreters? . . . . . . . . . . 65

4.2 Automatically Generated Threaded Interpreters . . . . . . . . . . 68

4.3 Limitations of the Procedural Representation . . . . . . . . . . 74

4.4 Overcoming Opacity . . . . . . . . . . 76

4.5 Summary . . . . . . . . . . 80

**5 Constructive Non-elementary Functions** **81**

5.1 Constructing functions from their Defining Properties . . . . . . . . . . 81

5.2 Construction by Abstraction and Composition . . . . . . . . . . 83

5.3 Closures are not Good Enough . . . . . . . . . . 85

5.4 Having Our Cake and Eating It Too . . . . . . . . . . 87

5.5 Summary . . . . . . . . . . 88

**6 Semantic Concerns** **91**

6.1 A Trivial Semantics for Translucent Procedures . . . . . . . . . . 91

6.2 Transparency Reveals Too Much . . . . . . . . . . 93

6.3 Translucency Reveals What We Can Use . . . . . . . . . . 94

6.4 Preliminary Concepts . . . . . . . . . . 95

6.5 Computable Canonicalization Functions . . . . . . . . . . 98

6.6 Summary . . . . . . . . . . 101

**7 Efficiency concerns** **103**

**8 Related and Further Work** **109**

8.1 Related Work . . . . . . . . . . 109

8.1.1 Reflection and Reification . . . . . . . . . . 109

8.1.2 Other Related Work . . . . . . . . . . 111

8.2 Further Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

**9 Conclusions 115**

**A Implementation details 119**

A.1 Implementation of TScheme . . . . . . . . . . . . . . . . . . . . . . . 119

A.2 Implementation of the Procedural Pattern Matcher . . . . . . . . . . . 121

A.2.1 Data Structures Manipulated by the Matcher . . . . . . . . . . 122

A.2.2 Comparison Walk in the Matcher . . . . . . . . . . . . . . . . . 123

A.2.3 Simple Match of Pattern Variables . . . . . . . . . . . . . . . . 126

A.2.4 Matching Combination Patterns . . . . . . . . . . . . . . . . . . 127

A.2.5 Under-constrained Values and Consistent Bindings . . . . . . 136

**B Denotational Semantics 141**

B.1 Semantics in Scheme notation . . . . . . . . . . . . . . . . . . . . . . 141

B.2 Semantics in Traditional notation . . . . . . . . . . . . . . . . . . . . 156

B.2.1 Abstract Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 156

B.2.2 Domain Equations . . . . . . . . . . . . . . . . . . . . . . . . . 156

B.2.3 Semantic Functions . . . . . . . . . . . . . . . . . . . . . . . . . 156

# Chapter 1

# Abstraction and Procedural Opacity

This report introduces *translucent procedures* as a new mechanism for implementing behavioral abstractions. Like an ordinary procedure, a translucent procedure can be invoked, and thus provides an obvious way to capture a *behavior*. Translucent procedures, like ordinary procedures, can be manipulated as first-class objects [58] and combined using functional composition. But unlike ordinary procedures, translucent procedures have structure that can be examined in well-specified non-destructive ways, without invoking the procedure.

I have developed an experimental implementation of a normal-order lambda-calculus evaluator augmented with novel reflection mechanisms for controlled violation of the opacity of procedures. I demonstrate the utility of translucent procedures by using this evaluator to develop large application examples from the domains of graphics, computer algebra, compiler design, and numerical analysis.

Abstraction is central to software design. In constructing complex software systems, we begin with some given building blocks, aggregate these using some available means of combination, and *abstract* the aggregates, i.e. characterize them in some economical way, so that we can think about them simply and incorporate them as building blocks in larger aggregates. One especially effective approach to abstraction is *behavioral abstraction.* This abstracts each building block as something that has a specified *behavior.* In using the building block, only the behavior should matter, not other aspects of its implementation. The methodology of abstract data types [40, 12] is one popular approach to behavioral abstraction. Object-oriented programming [12, 1] is another.

For languages that support procedures as first-class objects, representing building blocks as procedures is a particularly effective approach to behavioral abstraction [51, 1, 6]. A building block represented as a procedure has an obvious abstraction, namely, the procedure's input-output behavior. In addition, the system designer has a means of combination already at hand, namely, ordinary functional composition. Consequently, there are many examples of systems whose elegance and power derives from the use of procedural representations. This report will describe some of them.

On the other hand, implementing behavioral abstractions as ordinary procedures has limitations, or to put it more correctly, procedures do their job too well. Procedures are completely opaque structures. If an element is represented as a procedure, then the *only* way to interact with it is to invoke it. We cannot examine it, or inspect its internal structure in any way.

Working with objects represented as ordinary procedures is like packaging all items in identical opaque boxes. If we are handed a box that has a wick sticking out of it, we don't know whether the box contains a birthday cake or a bomb; and the only way to find out is to light the wick. A translucent procedure is a box that can be x-rayed. We can't actually open the box and disturb the contents, but we can get a look at what's inside.

In this chapter, we will motivate the use of translucent procedures with a simple example—an implementation in Scheme of the functional picture language developed by Peter Henderson [33]. Henderson's language illustrates both the elegance and the limitations of procedural representations. By introducing translucent procedures, we can retain the overall structure of the language, while overcoming the limitations.

## 1.1 Henderson's Picture Language

Henderson's language was motivated by designs with elements that are combined and replicated at different scales, such as M.C. Escher's woodcut *Square Limit.* (Fig. 1-2) The language consists of primitive, atomic pictures (or alternatively, of primitives for constructing atomic pictures from line segments and other geometric figures), and means of combination that superimpose, juxtapose, and rotate pictures. For instance, given pictures, `george`, `martha`, and `triangle`, a typical compound picture might be constructed as

```
(rotate90
 (beside (superpose george triangle)
         martha
         .6))
```

The resulting picture shows `george` superposed with `triangle`, side by side with `martha`, `george` and `triangle` taking up 60% of the figure, and the ensemble rotated 90 degrees counter-clockwise. (Fig. 1-3)

Pictures in Henderson's language do not have a fixed size, or even a fixed aspect ratio. Instead, a picture is always drawn relative to some specified rectangle, and the picture is drawn with its width and height scaled to match the rectangle. Thus, the different drawings shown in Fig. 1-4 are all the same picture, only drawn with respect to different rectangles.

This self-scaling property makes it convenient to create designs such as *Square Limit.* It also suggests a natural representation for a picture as a procedure that expects a rectangle as its argument and executes the relevant drawing operations. For example, a primitive picture that consists of a diagonal line drawn from the bottom left to the top right of the designated rectangle would be implemented as

```
(define diagonal-picture
```

```
(lambda (rectangle)
  (let* ((bot-left (origin rectangle))
         (top-right (+vect (horiz rectangle)
                           (+vect (vert rectangle)
                                  bot-left))))
    (draw-line (vect/x bot-left) (vect/y bot-left)
               (vect/x top-right) (vect/y top-right)))))
```

We assume here that a rectangle is a structure with an origin and two vectors, where the vectors represent the *bottom* and *left* edges of said rectangle. The `draw-line` procedure draws a line segment taking as arguments the horizontal ($x$) and vertical ($y$) coordinates of the beginning and end points of the segment.

The elegance of this procedural representation becomes apparent when we begin to implement the means of combination for pictures. The `superpose` combinator, which creates the superposition of two pictures, drawn in the same rectangle, is simply

```
(define (superpose pict1 pict2)
  (LAMBDA (rectangle)
    (begin
      (pict1 rectangle)
      (pict2 rectangle))))
```

To place one picture beside another in a rectangle, we split the rectangle into two subrectangles according to the specified ratio, then draw the first picture in the left subrectangle, and the second picture in the right subrectangle:

```
(define (beside left right ratio)
  (LAMBDA (rectangle)
    (let ((origin (origin rectangle))
          (horiz (horiz rectangle))
          (vert (vert rectangle)))
      (let ((delta (scale-vect horiz ratio)))
        (begin
          (left (make-rect origin
                           delta
                           vert))
          (right (make-rect (+vect origin delta)
                            (-vect horiz delta)
                            vert))))))))

(define (scale-vect vect factor)
```

```
  (make-vect (* factor (vect/x vect))
             (* factor (vect/y vect))))

(define (+vect vect1 vect2)
  (make-vect (+ (vect/x vect1) (vect/x vect2))
             (+ (vect/y vect1) (vect/y vect2))))

(define (-vect vect1 vect2)
  (make-vect (- (vect/x vect1) (vect/x vect2))
             (- (vect/y vect1) (vect/y vect2))))
```

Rotating a picture by 90 degrees amounts to drawing the picture in a rectangle that is rotated from the original rectangle by 90 degrees.

```
(define (rotate90 pict)
  (LAMBDA (rectangle)
    (let ((origin (origin rectangle))
          (horiz (horiz rectangle))
          (vert (vert rectangle)))
      (pict (+vect origin vert)
            (scale-vect vert -1)
            horiz))))
```

Other means of combination, such as `above`, which places one picture above another, can be defined similarly.

This procedural representation illustrates the power of behavioral abstraction. A compound picture need not know how its components were constructed, or even what these components are. It only needs to know how the components should be arranged relative to the rectangle given to the compound picture.

To better appreciate this point, consider how these operations would be written had we chosen a less abstract representation of pictures. The code in fig. 1-1 implements the same set of operations on pictures when the representation is a list of segments in the unit square.

In this list-of-segments representation, every means of combination must contain code to operate on every primitive picture element. The compound operations must manipulate the representation of the component pictures in order to scale and translate them appropriately.

```
(define (superpose pict1 pict2)
  (append pict1 pict2))

(define (beside left right ratio)
  (append
   (map (lambda (seg)
          (scale-x seg ratio))
        left)
   (map (let ((ratio* (- 1 ratio)))
          (lambda (seg)
            (shift-x (scale-x seg ratio*) ratio)))
        right)))

(define (rotate90 pict)
  (map (lambda (seg)
         (let ((rotate-point
                (lambda (point)
                  (point (point/y point)
                         (- 1 (point/x point))))))

           (segment
            (rotate-point (seg/start seg))
            (rotate-point (seg/end seg)))))
       pict))

(define (scale-x seg factor)
  (let ((start (seg/start seg))
        (end (seg/end seg)))
    (segment
     (point (* factor (point/x start))
            (point/y start))
     (point (* factor (point/x end))
            (point/y end)))))

(define (shift-x seg delta)
  (let ((start (seg/start seg))
        (end (seg/end seg)))
    (segment
     (point (+ (point/x start) delta)
            (point/y start))
     (point (+ (point/x end) delta)
            (point/y end)))))
```

Figure 1-1: List-of-segments representation of Henderson's pictures

The advantage of Henderson's more abstract procedural representation is significant. The means of combination in Henderson's language constitute a language for composition, for arranging pictures, not for detailed drawing. The procedural representation allows us to implement it as such, ignoring all details about how the actual primitive pictures will be drawn. All knowledge of the low-level details of drawing is hidden inside the component procedures, and the compound picture need do nothing but invoke the components on suitable fragments of its drawing area. At no level in the decomposition does a picture need to know how its components will be drawn or what they consist of, just how they will be arranged in the rectangle.

Consider what would happen if we were to add circular arcs to our repertoire of atomic picture elements. Virtually every procedure in the list-of-segments representation from fig. 1-1 be affected, while none of the code in the original version would need to be modified.

We note in passing another advantage of the procedural representation: Having chosen to implement pictures as procedures, we can build the geometric combinations with no other mechanism than functional composition. In contrast, even the simple list-of-segments representation requires list operations such as `append`. Using fancier data structures for pictures would require us to implement additional data-structure operations.

## 1.2 Limitations of Procedural Representations

Although the procedural representation in Henderson's language has clear advantages over alternate representations, it has one fundamental drawback. Procedures in Scheme are opaque. The only operation defined on a procedure is invocation on suitable arguments. Once we have represented a picture as a procedure, the only thing we can do to it is to draw it! Of course, we can choose where and how to draw it, and that is how the means of combination in Henderson's language work, but there are other operations on pictures that we cannot implement, precisely because we have hidden all of the details of a picture inside the procedure that draws it, and we cannot examine this procedure.

For example, we might want a predicate that tests whether a picture is the `beside` of two other pictures, or an operation to decompose the result of `beside` into its constituent pictures. Alternatively, we may want to know whether a picture would render anything in its upper right quadrant or whether a picture is blank. As we will see, we cannot construct such operations if pictures are implemented as opaque procedures.

With traditional representations, such as the picture as a collection of segments, we might have some difficulty answering such questions, but ultimately they are answerable. We can write arbitrarily complex recognizers to examine the detailed contents of a picture. Yet with the procedural representation, which is ideally suited to combination, we are stuck. Our representation allows us to ignore all details about the component pictures when writing the combinators, because the active procedural components take care of themselves. But our pictures have inherited another property of procedures, namely their opacity, which is not a desirable property for our pictures.

To overcome this problem, we could resort to a variety of unsavory tricks. Conceivably we could draw into a bit-map and then examine the resulting bits, but this is clearly unappealing, and imprecise, since the act of drawing into a bitmap loses information. Alternatively we could *switch* the graphics device driver with a fake device that collects the operations, but we would still be losing the hierarchical information that our procedure representation keeps internally—the decomposition of a picture used in building it has been thrown away in the process of drawing, and it would have to be rediscovered from scratch.

Alternatively, we might change the interface to our procedures that represent pictures. We could, for example, create an object-oriented message-passing-like implementation, in which a picture procedure receives a message that indicates the operation it should perform. It would then either draw, or return information about how it can be decomposed. Our primitive pictures and our means of combination might then be something like the following:

```
(define diagonal-picture
  (lambda (message)
    (case message
      ((DRAW)
       (lambda (rectangle)
         (let* ((bot-left (origin rectangle))
                (top-rite (+vect (horiz rectangle)
                                 (+vect (vert rectangle)
                                        bot-left))))
           (draw-line (vect/x bot-left) (vect/y bot-left)
                      (vect/x top-rite) (vect/y top-rite)))))
      ((DECOMPOSE)
       (list 'SEGMENT '(0 0) '(1 1)))
      (else
       (error "Unknown message" message)))))

(define (superpose pict1 pict2)
  (lambda (message)
    (case message
      ((DRAW)
       (lambda (rectangle)
         (begin
           ((pict1 'DRAW) rectangle)
           ((pict2 'DRAW) rectangle))))
      ((DECOMPOSE)
       ;; or
       ;; (list 'SUPERPOSE
       ;;       (pict1 'DECOMPOSE)
       ;;       (pict2 'DECOMPOSE))
       (list 'SUPERPOSE pict1 pict2))
      (else
       (error "Unknown message" message)))))
```

This approach is unsatisfactory for several reasons:

First, we have introduced another representation, that is composed simultaneously with our procedures, along with a way to translate between procedures and this alternate representation. Essentially, we have written two versions of `superpose`, although they are collected in a single object, and will have to do this for every means of combination. Although the additional code is neither difficult nor particularly unnatural in this example, simultaneously maintaining multiple representations can become both difficult and cumbersome. We have been driven to this only because our original representation does not permit inspection.

Second, we have obscured the basic representation of a picture to overcome a drawback in our language. We don't want to think of pictures as procedures that take messages and return either procedures to draw on rectangles or lists of identifiers. We want to think of them as procedures that draw on rectangles and that somehow, we can decompose.

## 1.3 Overcoming Opacity

Rather than give in to undesirable tricks, or maintain multiple representations, we can require instead that our language provide us with a way to inspect the structure of procedures. Once we do this, we will be able to represent our pictures, and more importantly, our picture operations, as originally intended, without surrendering the ability to examine the result. We call such an inspectable procedure a *translucent procedure*. In our proposed extension to Scheme, translucent procedures are constructed using exactly the same syntax as ordinary procedures, except using the keyword `tlambda` in place of `lambda`.

The following chapter will give a careful exposition of translucent procedures and `tlambda`. For now, think of a translucent procedure informally as a list structure, the body of the procedure, with substitutions implied by the environment bindings and $\beta$-reduction [7]. With the procedures available as lists, we can compare and destructure procedures, for example, to inspect the structure of pictures represented as procedures in the Henderson language.

Destructuring procedures element by element is cumbersome. Thus, we also implement a procedural pattern matcher that simplifies the task of examining `tlambda` structures. The following chapter will describe the procedural pattern matcher in detail, but for now, suffice it to say that its operation is primarily *structural.* Thus, the pattern matcher avoids having to solve arbitrary procedural (functional) equations, which is undecidable in general.

To use the pattern matcher, one invokes the procedure `match?` with two arguments. The first argument, the *pattern,* may be a procedure, a pattern variable, or the result of composing procedures and pattern variables.[1] The second argument, the *instance,* is the procedure to compare or destructure.

The matcher attempts to find values for the pattern variables such that when the values are substituted for the pattern variables, the pattern and instance are equal. `match?` returns *false* if it cannot make the pattern and instance equal. Otherwise it returns a dictionary pairing the pattern variables with the values (procedures or constants) that make the substituted pattern equal to the instance. The dictionary will be empty if the pattern contains no pattern variables. Note that the match may be ambiguous, that is, multiple sets of pattern variable bindings will satisfy the equation, but the pattern matcher returns only one set of bindings. The procedure `match/lookup` finds the binding for a pattern variable in a dictionary returned by `match?`.

Examples of patterns and the use of the matcher are:

```
(match? (tlambda (x) (+ (#?F x) #?G))
        (tlambda (a) (+ (* a (+ a 5)) 56)))
```

which succeeds with bindings

```
#?F ⇌ (tlambda (x) (* x (+ x 5)))
#?G ⇌ 56

(match? (compose (tlambda (x) (#?F (* x x)))
                 (tlambda (y) (* y (#?G (+ y 3)))))
        (tlambda (a)
          (let ((b (* a
```

[1] Pattern variables look like identifiers preceded by "`#?`". For example, `#?FOO` is a pattern variable.

```
                    (let ((c (+ a 3)))
                      (* c c)))))
         (+ (* b b) 8))))
```

which succeeds with bindings

```
#?F ⥤ (tlambda (v) (+ v 8))
#?G ⥤ (tlambda (v) (* v v))
```

## 1.4 The Picture Language with Translucent Procedures

Given translucent procedures and the procedural pattern matcher, it is straightforward to extend the picture language to examine the structure of pictures.

First we re-implement the picture language to use translucent procedures instead of ordinary procedures. This is simply a matter of using exactly the same code as in [section 1.1] above, writing `tlambda` in place of `lambda`. For instance, the primitive `diagonal` picture becomes

```
(define diagonal-picture
  (TLAMBDA (rectangle)
    (let* ((bot-left (origin rectangle))
           (top-right (+vect (horiz rectangle)
                             (+vect (vert rectangle)
                                    bot-left))))
      (draw-line (vect/x bot-left) (vect/y bot-left)
                 (vect/x top-right) (vect/y top-right)))))
```

and the `beside` combinator becomes

```
(define (beside left right ratio)
  (TLAMBDA (rectangle)
    (let ((origin (origin rectangle))
          (horiz (horiz rectangle))
          (vert (vert rectangle)))
      (let ((delta (scale-vect horiz ratio)))
        (begin
          (left (make-rect origin
                           delta
```

```
                              vert))
              (right (make-rect (+vect origin delta)
                               (-vect horiz delta)
                               vert)))))))
```

and so on for `rotate90`, `above`, and other combinators.

We can use the matcher to recognize whether, for example, a picture was generated by the `beside` combinator:

```
(define (beside? pict)
  (match? (beside #?LEFT #?RIGHT #?RATIO)
          pict))
```

More usefully, we can decompose a `beside` combination into components:

```
(define (decompose-beside pict)
  (let ((result (match? (beside #?LEFT #?RIGHT #?RATIO)
                        pict)))
    (and result
         (list (match/lookup result #?LEFT)
               (match/lookup result #?RIGHT)
               (match/lookup result #?RATIO)))))
```

We can write `decompose-above` in the same style, and we can write progressively more complex recognizers. For instance, if `square` and `triangle` are pictures, we could recognize a `house` as a triangle above a square:

```
(define (house? pict)
  (cond ((decompose-above pict)
         => (lambda (result)
              (and (match? triangle (car result))
                   (match? square (cadr result)))))
        (else
         false)))
```

We can transform pictures by extracting their components, and then reassemble them in other ways. The following `re-squish` operation decomposes an `above` picture and reassembles it using a new ratio:

```
(define (re-squish pict new-ratio)
  (let ((result (decompose-above pict)))
    (if (not result)
        pict
        (above (car result) (cadr result) new-ratio))))
```

We can even walk a picture all the way down to its primitive components and rebuild it with transformed components. For example, assume that primitive pictures are colored segments constructed by `segment->picture`, defined as follows:

```
(define (segment->picture color x0 y0 x1 y1)
  (TLAMBDA (rectangle)
    (let ((origin (rect/origin rectangle))
          (horiz (rect/horiz rectangle))
          (vert (rect/vert rectangle)))

      (let ((map-segment
             (lambda (x y)
               (+vect origin
                      (scale-vect horiz x)
                      (scale-vect vert y)))))

      (let ((start (map-segment x0 y0))
            (end (map-segment x1 y1)))

        (draw-colored-line color
                           (vect/x start) (vect/y start)
                           (vect/x end) (vect/y end)))))))
```

Given a segment, we can recover the color and coordinates by using the matcher:

```
(define (decompose-segment pict)
  (cond ((match? (segment->picture #?color #?x0 #?y0 #?x1 #?y1)
                 pict)
         => (lambda (dict)
              (list (match/lookup dict #?color)
                    (match/lookup dict #?x0)
                    (match/lookup dict #?y0)
                    (match/lookup dict #?x1)
                    (match/lookup dict #?y1))))
        (else
         false)))
```

Now we can define an operation that recolors a picture given a color map.

```
(define (recolor pict color-map)
  (cond ((decompose-above pict)
         => (lambda (match)
              (above (recolor (car match) color-map)
                     (recolor (cadr match) color-map)
                     (caddr match))))
```

```
          ((decompose-beside pict)
           => (lambda (match)
                (beside (recolor (car match) color-map)
                        (recolor (cadr match) color-map)
                         (caddr pict))))
          ...
          ((decompose-segment pict)
           => (lambda (match)
                (apply segment->picture
                       (cons (color-map (car match))
                             (cdr match)))))
          (else
           (error "recolor: Unrecognized picture"
                  pict))))

(recolor house
          (lambda (color)
            (if (color=? color pink)
                blue
                color)))
```

## 1.5 Summary

The picture language is a simple toy example, yet it illustrates both the power and the limitations of procedural representations. With translucent procedures, we can have our cake, and eat it, too. We can continue to use procedures as the primitive data type for constructing pictures. Yet, we can also decompose pictures and examine them in situations where drawing is not the only interesting operation.

In the next chapter, we give a careful description of translucent procedures and the matcher. We consider some more significant uses of translucent procedures—building equation solvers, constructing interpreters and compilers, and generating mathematical libraries from mix-and-match components. Like the picture language, each of these applications could make elegant use of procedural representations, were it not for the limitations of opacity. In each case, we show how translucent procedures permit us to maintain the benefits of procedural representations, while overcoming the limitations of opacity. After discussing these examples, we return to some general considerations about the semantics of translucent procedures, efficiency of the implementation, and comparisons with other work.

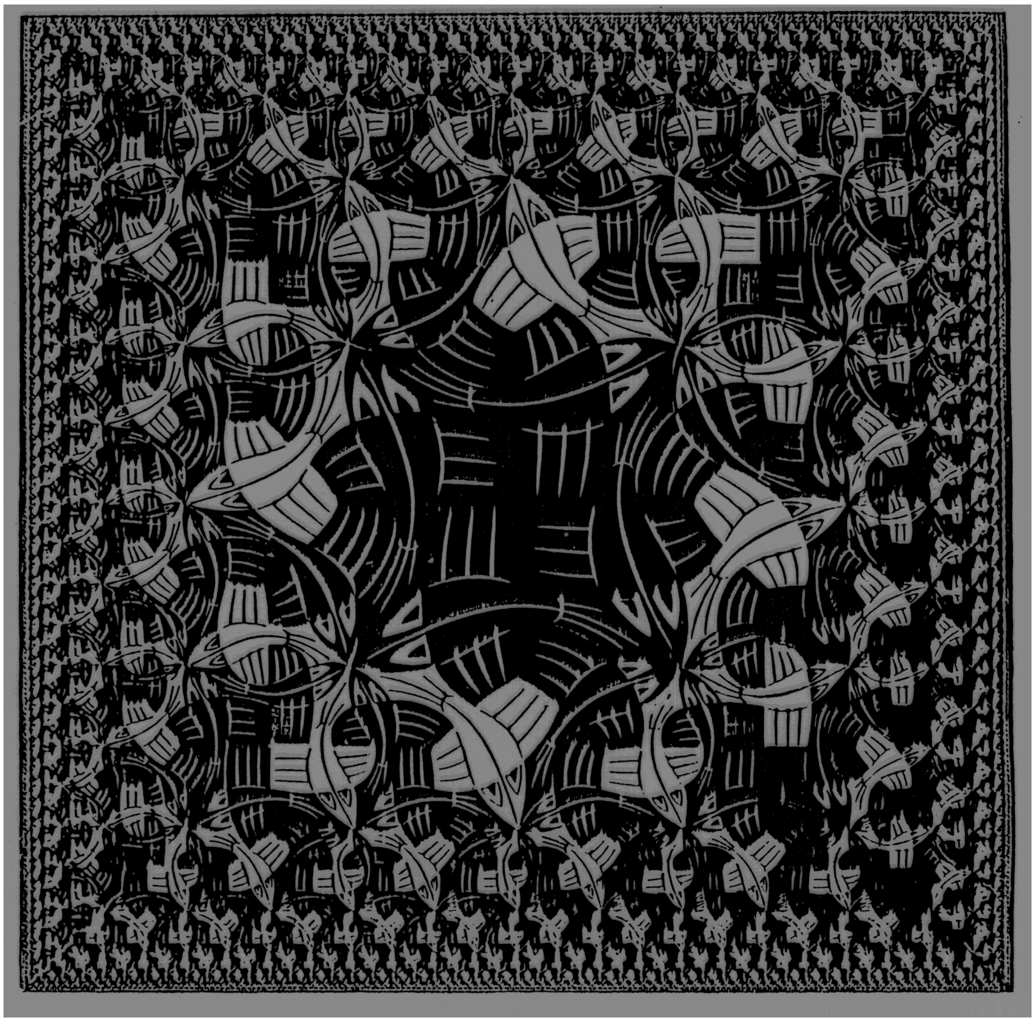

Figure 1-2: M.C.Escher's *Square Limit*

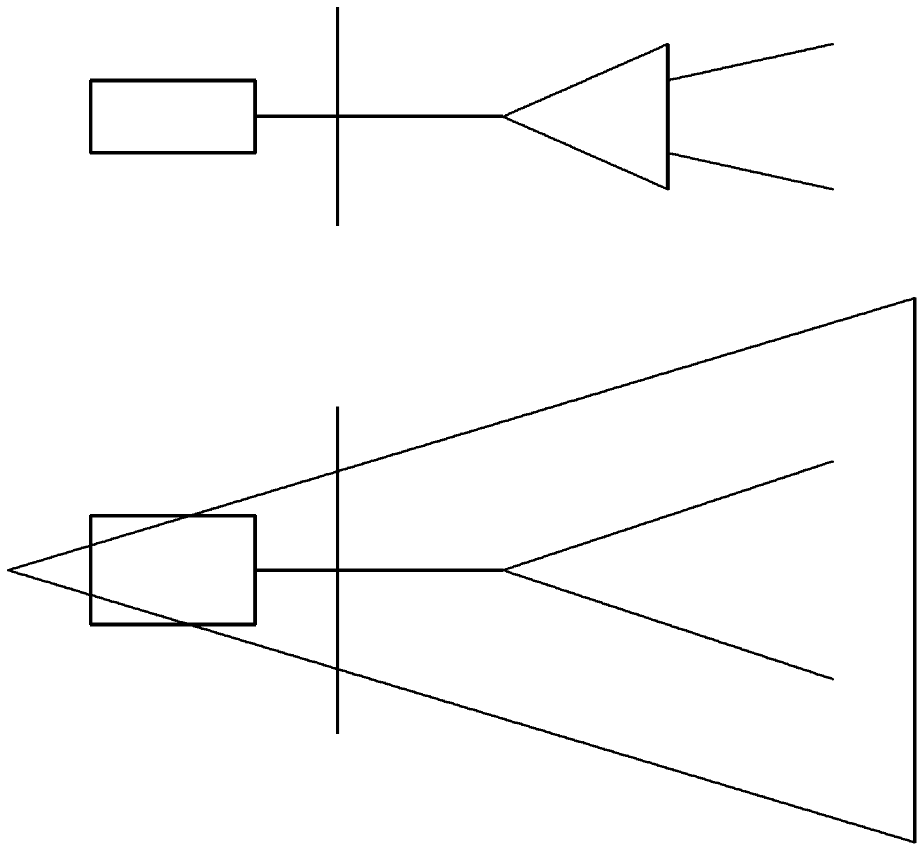

Figure 1-3: `george`, `triangle`, and `martha`

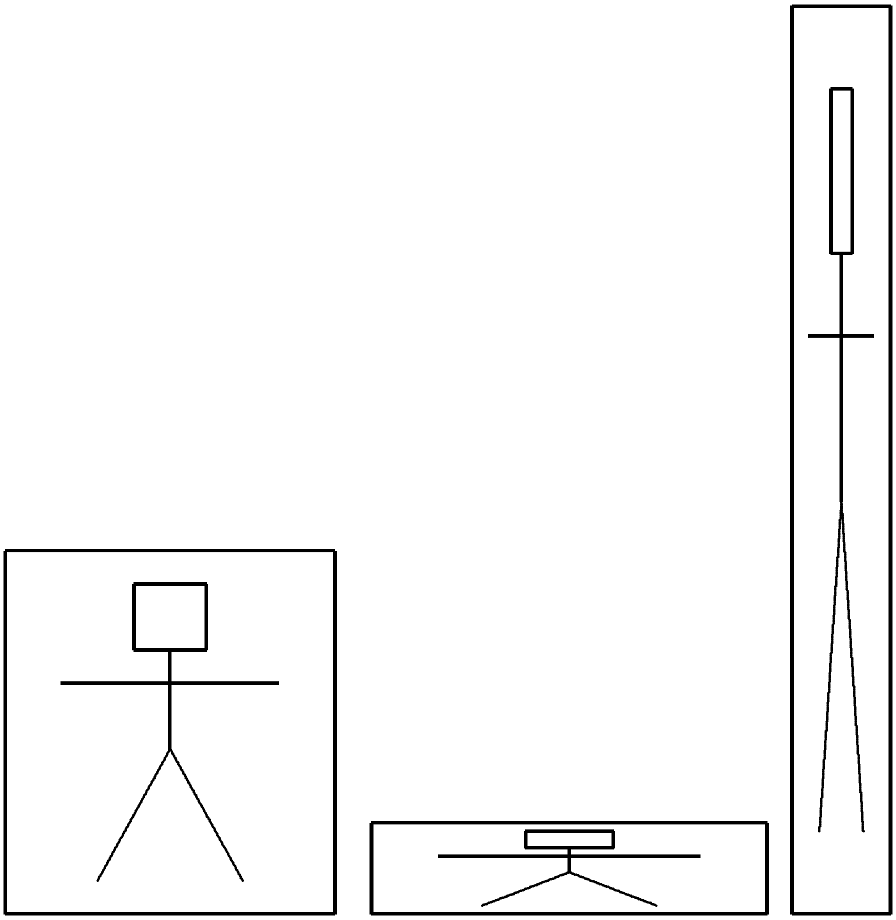

Figure 1-4: `george` in three different rectangles

# Chapter 2

# Translucent Procedures and the Procedural Pattern Matcher

The preceding chapter glosses over two significant issues:

- How `tlambda` differs from `lambda`;
- What the procedural pattern matcher does, exactly.

This chapter addresses these issues.

## 2.1 `tlambda` and TScheme

To experiment with non-opaque procedures, and with tools that manipulate procedures by operating on their expressions and environments, we will use a Scheme-like language called TScheme. Using a new language allows us to change the semantics, and in particular, to experiment with procedure representations, at will. We cannot use Scheme directly because it only has opaque procedures.[1] However, to avoid pointlessly duplicating existing utilities, we can embed TScheme in Scheme, instead of implementing it from scratch.

[1] MIT Scheme procedures are not opaque, but their reified representation is cumbersome at best.

TScheme has special forms with the same names as those present in ordinary Scheme. In particular, it has a `lambda` special form used to create procedures. Since Scheme and TScheme special forms overlap, we need a way to declare that part of a program is written in TScheme instead of Scheme. `tlambda` is the escape mechanism from Scheme to TScheme. It is a simple Scheme macro that constructs a TScheme procedure whose body will be evaluated using the TScheme interpreter. For example, the expression

```
(let ((z (sqrt 2)))
  (tlambda (x y)
    (let ((w (+ z z)))
      (+ (* x w) y))))
```

is essentially rewritten as

```
(let ((z (sqrt 2)))
  (tproc/make `(lambda (x y)
                 (let ((w (+ z z)))
                   (+ (* x w) y)))
              `((+ ,+) (z ,z) (* ,*))))
```

where `tproc/make`, described below, constructs a procedure whose body is evaluated using TScheme semantics.

The programs in this report are written in an amalgam of Scheme and TScheme, with `tlambda` expressions marking the transition points. Code surrounding `tlambda` expressions is written in Scheme, while the code inside these expressions is written in TScheme. For example, in the expressions above, the outer `let` expressions are Scheme `let` expressions, while the inner ones are TScheme `let` expressions.

The examples are written in this amalgam for convenience, and also to point out exactly which procedures need to be translucent, but there is no a-priori reason why the whole code could not be written in TScheme.

TScheme differs from Scheme in both fundamental and accidental ways.

### 2.1.1 Fundamental Differences

#### Reflective and Reifying Operations

The most fundamental difference, obviously, is that TScheme procedures are not opaque. Three reflective and reifying operations [24], available in both Scheme and TScheme, provide the primitive manipulation ability of TScheme procedures, abbreviated *tprocs*. These operations suffice to build the higher-level pattern matcher.

- `(tproc?` *object*`)` $\rightarrow$ *boolean*

  `tproc?` returns *true* if *object* is a TScheme procedure (created by `tlambda` or a TScheme `lambda`), *false* otherwise. As a result of the embedding, all TScheme procedures also answer *true* to `procedure?`.

- `(tproc/make` *expr env*`)` $\rightarrow$ *tproc*

  `tproc/make` returns the TScheme procedure that results from evaluating TScheme `lambda` expression *expr* (an S-expression) in a TScheme environment whose bindings correspond to the pairs in the list *env*. *env* is a list of pairs of variable names (symbols) and arbitrary values. *env* must contain bindings for all the free variables of *expr*.

- `(tproc/decompose` *tproc*`)` $\rightarrow$ *expr * env*

  `tproc/decompose` is an inverse of `tproc/make`. It returns two values, a TScheme `lambda` expression (as an S-expression), and an association list (alist) representing the environment of *tproc*.

`tproc/make` and `tproc/decompose` are not exact inverses. Invoking `tproc/make` on the result of `tproc/decompose` produces a TScheme procedure distinguishable from the original only by `eq?` and `eqv?`. On the other hand, invoking `tproc/decompose` on the result of `tproc/make` may produce a different expression and environment from those given to `tproc/make`. The expression may have some of its variables renamed, and various substitutions may have been performed, e.g. `let` expressions may have been transformed into equivalent combinations with `lambda` expressions as the operators. The environment may have its bindings in a different order, it may have some of the variables renamed to agree with renamings in the expression, it will have unreferenced variables removed, and may have additional variables needed by some of the substituted expressions. For example,

```
(tproc/decompose
 (tproc/make '(lambda (x)
               (+ x z))
            `((+ ,+) (z ,7) (* ,*) (w ,4))))
```

may return

```
(lambda (y)
  (+ y 7))
```

*and*

```
((+ <primitive +>))
```

`tproc?` is used by programs to discriminate between TScheme procedures and ordinary Scheme procedures. `tproc/decompose` is used only in the implementation of the matcher. `tproc/make` is used directly in the expansion of `tlambda`, and indirectly through the use of `tproc/make*`, defined as follows.

```
(define (tproc/make* lam-expr)
  ;; Empty environment.
  ;; LAM-EXPR should have no free variables.
  (tproc/make lam-expr '()))
```

### Multiple Values

An additional important difference between TScheme and Scheme is that the number of arguments and return values of TScheme procedures are fixed, and a *syntactic* property of the `lambda` expression that produced the procedure. TScheme `lambda` expressions specify a fixed number of arguments, i.e., no optionals, *dot* notation, or `&rest` [11, 57]. Consequently, the number of arguments of a TScheme procedure can be obtained using the `tproc/arity` procedure. The number of values returned by a TScheme procedure can be obtained using the `tproc/nvalues` procedure.[2]

Multiple return values are declared, at the lowest level, by using the `tuple` special form. When evaluated, it returns as many values as it has operands. Multiple values are bound to variables by using the `tlet` special form. `tlet` is similar to `let`, but binds several variables to the values returned from a single expression. For example, the following two procedures compute the same values:

```
(tlambda (a b)
  (tlet ((x y) (tuple (+ a b) (- a b)))
    (* x y)))

(tlambda (a b)
  (let ((x (+ a b))
        (y (- a b)))
    (* x y)))
```

In addition to `tuple` and `tlet`, the following two assumptions are necessary to be able to compute the number of values returned by a TScheme expression:

- Both branches of a conditional must return the same number of values.
- Unknown procedures (e.g. parameters or free variables) return exactly one value unless they are directly called in such a position that the values returned by them would be bound to the variables in a `tlet` expression. In this case, the number of returned values must be the number of variables bound by the `tlet` expression.

As an example of the use of several of these peculiar operators on TScheme procedures, consider the following definition of `compose`:

[2] `tproc/nvalues` and `tproc/arity` are provided primitively even though they can be written using `tproc/decompose`.

```
(define (compose f g)
  ;; h = (compose f g)
  ;; => (h . args) = (f . (g . args))
  (let ((ftakes (tproc/arity f))
        (freturns (tproc/nvalues f))
        (gtakes (tproc/arity g))
        (greturns (tproc/nvalues g)))

    (if (not (= ftakes greturns))
        (error "compose: Incompatible" f g)
        (let ((names (make-list-of-names
                       (+ gtakes greturns freturns)
                       '(f g))))
          (let ((formals (list-head names gtakes))
                (rest (list-tail names gtakes)))
            (let ((middle (list-head rest greturns))
                  (results (list-tail rest greturns)))
              ((tproc/make*
                `(lambda (f g)
                   (lambda ,formals
                     (tlet (,middle (g ,@formals))
                       (tlet (,results (f ,@middle))
                         (tuple ,@results))))))
               f g)))))))
```

`make-list-of-names` takes an integer $n$, and a list of symbols $l$, and returns a list of symbols of length $n$ with no duplicates and whose intersection (as a set) with $l$ is empty. `list-head` and `list-tail` take a list $l$, and an integer $n$, and return the initial (or final, respectively) sublists of $l$ when split before the $n$-th element.

### 2.1.2 Accidental Differences

In addition to the essential differences described earlier, there are some inessential differences as well. These differences were introduced in order to make some simplifying assumptions when constructing TScheme and the procedural pattern matcher. Although these differences can be eliminated, their consequences are notable. These accidental differences are:

1. Scheme is a *call-by-value* language. TScheme is a *call-by-need* language [25]. In Scheme, arguments to a procedure are evaluated fully before the procedure is entered. In TScheme, the evaluation of arguments is delayed until the called procedure needs their values. Arguments are evaluated fully only when needed by the called procedure, and their values memoized to prevent potentially costly re-evaluation.

   The reason for this difference is that it allows programs to substitute argument expressions for formal parameters in procedure calls without worrying about termination or errors.

   To illustrate this point, consider the following expressions.

```
(lambda (x)
  (let ((y (foo x)))
    (if (bar? x)
        y
        3)))

(lambda (y)
  (if (bar? x)
      (foo x)
      3))
```

   When viewed as Scheme programs, they are not identical in behavior. The first may not terminate or may signal a runtime error in cases when the second terminates normally. By contrast, when considered as TScheme programs, the above two expressions produce procedures that behave identically, since the call to `foo` will only be executed if `(bar? x)` is true, in either case. Thus the substitution of `y` by `(foo x)` in the code yields equivalent programs in TScheme but not in Scheme.

   If we are to change TScheme to be a call-by-value language, we either have to depend on some approximate strictness analysis before performing the substitution, or, alternatively, unilaterally declare that termination and error properties are not properly preserved by our tools. The latter choice may appear extreme, but is sometimes used when writing compilers [3, Alternative Code Motion Stragegies, Dead-Code elimination].

One minor consequence of the call-by-need semantics of TScheme is the peculiar behavior of the `begin` special form; It evaluates all of its operands fully in left-to-right order and returns the value of the rightmost. In Scheme the following two programs are equivalent:

```
(begin
  (draw-line 0 0 1 1)
  (draw-line 1 1 1 0))

(let ((first (draw-line 0 0 1 1))
      (rest (lambda () (draw-line 1 1 1 0))))
  (rest))
```

However, these two programs are not equivalent in TScheme. The first draws two lines. The second draws only one, because the variable `first` is bound to a delayed evaluation that is never needed by the body of the `let` expression. However, the following program would have the same effect as the first in both Scheme and TScheme:

```
(let ((first (draw-line 0 0 1 1))
      (rest (lambda () (draw-line 1 1 1 0))))
  (begin
    first
    (rest)))
```

2. TScheme does not have a `set!` special form; Variables are immutable in T-Scheme. There are two reasons for this:

   - Mutation and call-by-need languages do not mix very well. Because mutation causes the values of expressions to depend on time, and call-by-need languages make the time of evaluation of expressions hard to predict, it is difficult to construct programs in the presence of both.

- Variable assignment complicates environments. In a language without assignment, an environment is simply a function mapping variables to values. Such functions, being constant, can be projected, and the value returned by them is independent of how and when other expressions are evaluated. In a language with assignment, the values of variables may not be constant over time, and the recognition of whether a variable is affected is not local. For example, in Scheme, the decision to substitute the value of `x` into the `lambda` expression

```
(lambda (y)
  (+ x y))
```

  depends on what other code shares the same binding of `x`. We can substitute it if the complete code is the first of the following, but not if it is the second:

```
(lambda (x)
  (lambda (y)
    (+ x y)))

(lambda (recv)
  (let ((x 0))
    (recv (lambda (x*) (set! x x*))
          (lambda (y)
            (+ x y)))))
```

  In the absence of variable assignment, environments can be merged and manipulated more easily.

Assignment can be added to TScheme if we add appropriate declarations, or if only complete programs are manipulated.

3. There are no *forward references* in TScheme. All free variables of a TScheme `lambda` (or a Scheme `tlambda`) expression must be bound, and their values available, when the `lambda` (or `tlambda`) expression is evaluated.

   Consider the following program:

```
(lambda (x)
  (+ (foo x)
     (sqrt x)))
```

It is a legal Scheme program in the absence of a previous definition of `foo`. Calls to the resulting procedure, however, must be postponed until `foo` has been defined. By contrast, it is a legal TScheme program only if `foo` has previously been defined.

The lack of forward references simplifies the code for the TScheme evaluator, since there are no unbound variables in TScheme. No variable substitutions need to be delayed because the value is not yet available, and no mechanism for delaying the reference until the value is available needs to be implemented.

It is not hard, merely cumbersome, to allow TScheme programs to resolve some variable references later.

Of course, the lack of forward references does not preclude recursion, since the fixed-point combinator $Y$ [58, 7] is expressible in the language.

4. There is no `call-with-current-continuation` procedure in TScheme. Like variable assignment, `call-with-current-continuation` and call-by-need languages do not mix particularly well. The continuation in effect when an expression is evaluated is difficult to predict when the time, and, more importantly, the context, of that evaluation are hard to predict.

   If TScheme is changed to be call-by-value, or escape procedures are added anyway, it is not be hard to decompose continuations as well. For example, consider the following fragment, assuming call-by-value evaluation:

   ```
   (let ((expr
          (call-with-current-continuation
           (lambda (cont)
             ⟨body1⟩))))
     ⟨body2⟩)
   ```

   The value of `cont` might decompose into the following expression

```
(lambda (%val)
  (with-continuation K
    (lambda ()
      (let ((expr %val))
        ⟨body2⟩)))))
```

and an environment that contains bindings for the free variables of ⟨*body*2⟩ (except for `expr`), and `k` which is the rest of the continuation, another escape procedure. This code uses `with-continuation`, a procedure that takes two arguments, an escape procedure, and a *thunk*, i.e. a procedure of no arguments. It invokes the thunk with an implicit continuation corresponding to the escape procedure. [3]

## 2.2 The Procedural Pattern Matcher

`tproc/decompose` is very cumbersome to use, although sufficiently powerful. The procedural matcher is often adequate, and makes the code that examines and destructures procedures considerably simpler. The remaining chapters use the matcher exclusively, and before we proceed, we should describe it further.

As mentioned in the previous chapter, the pattern matcher `match?` takes two procedures as arguments. The first, the pattern, is a TScheme procedure that can use pattern variables. The second, called the instance because it is presumably an instantiation of the pattern, is an ordinary TScheme procedure. Pattern variables are recognizable uninterpreted constants. The values of the following expressions are valid arguments to `match?`:

```
(tlambda (x y) (+ (* x x) (* x y)))

(tlambda (x y) (+ (#?FOO x) (#?BAR y)))

(tlambda (x) (* x #?BAZ))
```

[3]Continuations are modelled syntactically in [22], which also models assignment. However, the operations used to model assignment are global, and unsuitable here. Fortunately, locations are a satisfactory representation if we are to handle assignment as well.

```
(let ((op #?FOO))
  (tlambda (z w)
    (+ (op z)
       (op w))))
```

As can be seen from the above examples, the first argument to `match?`, always a procedure, will often signal an error if invoked, since pattern variables are otherwise uninterpreted. The previous example also shows that the pattern variables need not appear directly in a TScheme expression, but may be introduced instead through its free variables. The matcher is not simply an expression pattern matcher, since the values of free variables are examined.

The procedural matcher can be used to compare procedures, and to solve simple procedural substitutional equations. The values that it computes for pattern variables are such that when substituted for the corresponding pattern variables, they make the pattern equivalent to the instance. Consequently, to understand what the matcher does, we need to understand its notion procedure equality.

When dealing with procedures, the most desirable notion of equality is behavioral equality, but this property is undecidable. Any useful notion of equality, however, should be a conservative approximation of this undecidable property. By conservative we mean that our notion of equality should consider procedures to be equal only if they are behaviorally equal, i.e., there should be no false positives.

One of the simplest non-trivial conservative notions of equality that we can use is *equality of appearance*. Under this definition, we consider two procedures equal if `tproc/decompose` produces two equal `lambda` expressions, and two environments binding the same names to identical values. This is, unfortunately, a patently unsatisfactory notion. Trivial variable renamings such as

`(tlambda (x) (* x x))` $\Leftrightarrow$ `(tlambda (y) (* y y))`

make otherwise identical procedures distinct.

More importantly, the `lambda` expression and environment provided by `tproc/decompose` are mostly a consequence of the history of the construction of a procedure, and not a property of the result per se. In the following example, `foo` and `bar` are considered different when using this notion of equality.

```
(define (*fcn f g)
  (tlambda (x)
    (* (f x) (g x))))

(define foo
  (tlambda (x) (* x x)))

(define bar
  (*fcn (tlambda (x) x)
        (tlambda (x) x)))
```

We can easily see that `foo` and `bar` are equal under a sufficient number of unfoldments ($\beta$-reductions), and we might consider our notion of equality to be *equality of appearance,* after an arbitrary number of unfoldments, but this notion is problematic:

- It is quite conservative. The following procedures that compute Fibonacci numbers cannot be made to appear equal under arbitrary unfoldment. They are not behaviorally equal either since they produce different values for negative and non-integer numbers, and one might run out of storage when the other would not. However, we can wrap them with code that to make the ensembles behaviorally equal, yet the ensembles will not appear equal under arbitrary unfoldment.[4]

  ```
  (define (rfib n)
    (if (< n 2)
        n
        (+ (rfib (- n 1)) (rfib (- n 2)))))

  (define (ifib n)
    (define (inner i fi fi+1)
      (if (>= i n)
          fi
          (inner (1+ i) fi+1 (+ fi fi+1))))
    (inner 0 0 1))
  ```

[4]Except, perhaps, trivial ones that do not use them.

- It is not decidable. The equality of higher-order function schemas with uninterpreted constants under infinite unfoldment is undecidable. Even if we restrict ourselves to a first-order subset, we will not be safe, since the first-order version of this problem, with uninterpreted constants, is not known to be decidable [26].

The first problem is not terribly disappointing. After all, it arises mostly due to additional axioms introduced by our primitives (e.g. arithmetic), and to fully account for it, we need a theorem prover powerful enough to solve all questions in the theory introduced by our primitives. For example,

```
(tlambda (n-2 x y z)
  (let ((n (+ n-2 2)))
    (= (expt z n)
       (+ (expt x n) (expt y n)))))
```

and

```
(tlambda (n-2 x y z)
  false)
```

are behaviorally identical for non-negative integers if and only if Fermat's Last Theorem is true, but this problem has been open for over three hundred years. Of course, even if Fermat's Last Theorem is ever proved or disproved, no complete finite axiomatization of arithmetic exists (Gödel's incompleteness result [17]), so it is not unreasonable to either give up additional equality theorems provided by our primitives, or to approximate those conservatively as well.

The second problem above is somewhat more distressing, but open to simple and useful approximations. For example, in the Henderson picture language example, and in the examples that we will explore later, a finite number of unfoldments suffices to unfold the procedures completely. We never need to unfold a procedure again while in the process of unfolding it—our procedures have no cycles, they are shaped as directed acyclic graphs (DAGs). This condition is easily checked, and our conservative equality tester can fail, i.e. return *false*, when it finds a cycle.

We can now state a useful very conservative equality condition:

A procedure is fully unfolded if no combination (procedure call) has a `lambda` expression in its operator position. Typically, operators will be primitives, lambda-bound variables, or other such combinations.

A procedure $X$ can be finitely fully unfolded if there is no procedure $Y$ that we need to unfold to unfold $X$ such that $Y$ needs to be unfolded to unfold $Y$ fully. In other words, $X$ can be finitely fully unfolded if, while unfolding it, we do not run across a recursive procedure.

For example, the following cannot be finitely unfolded fully because we need to unfold $\texttt{lambda}_3$ while unfolding it.

```
(lambda₁ (f)
  ((lambda₂ (x)
     (f (x x)))
   (lambda₃ (x)
     (f (x x)))))
```

Two procedures are considered equal if they are identical in the sense of `eq?` or if they both can be finitely fully unfolded, and after fully unfolding, the resulting expressions are equal except for arbitrary renamings of bound variables, i.e., $\alpha$-conversion [10, 7].

Of course, we can make our equality notion somewhat sharper by allowing identical (i.e. `eq?`) recursive procedures to be met at the same place in the unfolding of both procedures, but this is not necessary for the examples in this report.

This notion of equality is consistent with both strict (applicative order) and non-strict (normal order) languages. Consider the following program fragments:

```
(let ((x ⊥))
  ;; X is intentionally not referenced
  (if (fermat? 3 y z w)
      w
      0))

(if (fermat? 3 y z w)
    w
```

```
    0)
```

They are equivalent in non-strict languages, but not in strict languages. In non-strict languages, the unfoldment of the expression for the value of `x` can wait until `x` is referenced. In strict languages this expression has to be finitely unfolded *before* proceeding with the body. Thus, in a non-strict language, $\perp$ is never unfolded, and the expressions will match. In a strict language, $\perp$ is unfolded when the `let` is processed, and our equality predicate will fail.

Given this notion of equality, we can now describe the behavior of the matcher. When the pattern argument does not involve pattern variables, it is an equality tester for the conservative equality condition that we have described.

However, if the pattern procedure contains pattern variables, the procedural matcher attempts to compute substitutions for the pattern variables such that after performing those substitutions on the pattern, the substituted pattern and the instance procedure are equal according to our definition of equality. The matcher computes the substitutions by examining the subtree of the unfolded instance corresponding to the unfolded expression in the pattern where the pattern variable appears.

The pattern matcher returns false or a dictionary. It returns false when it cannot make the pattern and instance equal, otherwise it returns a dictionary. The dictionary binds the pattern variables present in the pattern to objects, procedures or constants, that satisfy the equality equation. If there are no pattern variables in the pattern, but the match succeeds, the returned dictionary is empty.

The operation and implementation of the matcher is described in detail in appendix A, but for now, we will become acquainted with its behavior primarily through examples.

Pattern variables can represent procedures or constants in the code. Pattern variables in non-operator position will match either constants or procedures at the corresponding place in the instance. Pattern variables in operator position will only match procedures limited by the structure of the pattern. For example,

```
(match? (tlambda (y) (+ y #?C))
        (tlambda (x) (+ x 3)))
```

will match with bindings

```
#?C ⇌ 3
```

while

```
(match? (tlambda (y) (+ y #?C))
        (tlambda (x) (+ x (* x 42))))
```

does not match because there is no constant or procedure binding for `#?C` that will make the pattern and instance equal. However,

```
(match? (tlambda (y) (+ y (#?P y)))
        (tlambda (x) (+ x (* x 42))))
```

will match with bindings

```
#?P ⇌ (tlambda (y) (* y 42))
```

When a pattern variable is used as the operator of a simple combination, the corresponding value is limited to procedures that take as many arguments as provided to the pattern variable. In addition, the pattern matcher will not create lambda expressions not present in the instance or implied by the combination containing pattern variables in operator position. For example,

```
(match? (tlambda (x y z) (#?F x (#?B y z)))
        (tlambda (x y z) (+ (- (* y y) z) x)))
```

will succeed with bindings

```
#?F ⇌ (tlambda (x b) (+ b x))
#?B ⇌ (tlambda (y z) (- (* y y) z))
```

while

```
(match? (tlambda (x y z) (#?F x (#?B y z)))
        (tlambda (x y z) (+ (- (* y y) x) z)))
```

will fail. Ignoring arithmetic rearrangement (the constant functions `+`, `-`, and `*` are uninterpreted), it needs bindings such as

```
#?F ⇌ (tlambda (x b) (b x))
#?B ⇌ (tlambda (y z)
        (lambda (x)
          (+ (- (* y y) x) z)))
```

where the inner `lambda` expression in the binding for `#?B` neither appears in the instance, nor is implied by the pattern, in which the nesting of the pattern variable `#?B` implies only one `lambda` expression.

The number of arguments, and the nesting of the binding of a pattern variable that appears in operator position is restricted by the structure of the pattern. For example, the pattern

```
(tlambda (x y z)
  ((#?F x z) y))
```

restricts the binding for `#?F` to have the following shape:

```
(tlambda (a b)
  (lambda (c) ⟨body⟩))
```

where ⟨*body*⟩ can only be a `lambda` expression that corresponds to a `lambda` expression explicitly appearing in the unfolded instance.

This constraint on the values of the pattern variables used as operators of combinations depends on the context of the combination. For example, both the previous example and the following patterns

```
(tlambda (x z)
  (#?F x z))

(tlambda (x y z w t)
  (((#?F x z) y) w t))
```

contain the same application, whose operator is the pattern variable `#?F`, but the latter two patterns impose different restrictions on the binding for `#?F`, namely,

```
(tlambda (a b) ⟨body1⟩))

(tlambda (a b) (lambda (c) (lambda (d e) ⟨body2⟩)))
```

respectively. Again, ⟨*body*1⟩ and ⟨*body*2⟩ can only be `lambda` expressions if they correspond to `lambda` expressions that appear in the instance.

This context dependence of patterns may seem arbitrary, but is a natural consequence of the process of comparing by unfolding. In order to unfold a combination, we need to unfold the operator first, until it yields a primitive or a `lambda` expression. Any combination whose operator is a combination cannot be unfolded until its operator is unfolded, thus we need to find the innermost operator first. If we find a pattern variable as the innermost operator, we cannot unfold any of the surrounding combinations, and we must attempt to bind the pattern variable, considering the operands accumulated at all levels as the curried arguments to the binding of the pattern variable. Thus, two patterns that differ only in the signature of the pattern variable will match the same procedures, but with bindings that differ only in their signature. In other words, the signature for the value of a pattern variable is syntactically determined by the way the pattern variable is used.

For example, if a pattern contains the combination `(#?F x y z)` as the only reference to pattern variable `#?F`, and matches some instance with the following binding:

```
#?F ⇌ (tlambda (x y z) (* y (+ x z)))
```

mergepar Then the same pattern, with `((#?F x y) z)` replacing `(#?F x y z)`, will also match that instance, but with the following binding instead:

```
#?F ⇌ (tlambda (x y)
        (tlambda (z) (* y (+ x z))))
```

Of course, if both `(#?F x y z)` and `((#?F x y) z)` appear in the same overall pattern, the matcher will fail, since the constraints imposed on the signature for the binding of `#?F` are inconsistent.

One must keep in mind that although we will use the pattern matcher exclusively in the rest of this report, it is just an example of the tools that can be built once the opacity of procedures is abandoned. The matcher is sufficient for the problems at hand, but perhaps not for others. More powerful tools, such as unifiers, can be built as well. Of course, when doing so, one must keep in mind that many intuitively clear processes on expressions are undecidable, such as higher-order unification.

Now that we understand our language and tools better, we can proceed to apply them to other scenarios.

# Chapter 3

# Solvers for Systems of Equations

With our new set of tools, we can explore additional scenarios where using procedures as primitive data elements, and functional composition as glue, leads to elegant solutions. In all of these scenarios opaque procedures ultimately prevent us from using a procedural representation, while translucent procedures and the procedural matcher permit elegant solutions.

In this chapter we will see how functional abstraction and composition can be used to write an equation solver without the need for expression substitution and variable renaming. Opaque procedures are an unsuitable representation for equations because they do not permit inspection, necessary to choose the solution method to use. Translucent procedures enable this choice and preserve the elegant structure of the solver.

## 3.1 Solving Systems by Substitution

Consider a system of linear equations:

$$\begin{align} 2x - y + z &= 5 \tag{3.1} \\ x + y + z &= 6 \tag{3.2} \\ x - y + z &= 2 \tag{3.3} \end{align}$$

The simplest method for solving systems of linear equations, although not the most efficient, is to use substitution.[1] Using one of the equations, we solve for one of the unknowns in terms of the rest, and then substitute the resulting expression for the solved unknown in the remaining equations, reducing the number of equations and the number of unknowns. We then proceed to solve the residual system of rewritten equations, and once the values of the rest of the unknowns are known, we can compute the value of the eliminated unknown by using the same substitution expression.

In our example, we might use equation 3.1 to solve for $x$ in terms of $y$ and $z$,

$$x = ((5-(-y+z))/2) = ((y+5-z)/2) \tag{3.4}$$

and substitute this expression into equations 3.2 and 3.3 to produce the following reduced system after simplification:

$$\begin{aligned} 3y+z &= 7 && (3.5)\\ -y+z &= -1 && (3.6) \end{aligned}$$

We repeat the process with this system, solving for $z$ by using equation 3.6:

$$z = ((-1-(-y))/1) = y-1 \tag{3.7}$$

and substituting into equation 3.5 to obtain, after simplification

$$4y = 8 \tag{3.8}$$

which we can solve using the same linear elimination process:

$$y = 8/4 = 2$$

Now we can find the values for $z$ and $x$ by *plugging in*, that is, by using the previously computed substitutions.

$$\begin{aligned} &z = (2-1) = 1 && \text{from eqn. 3.7}\\ &x = ((5+2-1)/2) = 3 && \text{from eqn. 3.4} \end{aligned}$$

The substitution method consists of four steps:

[1] Substitution as described here and when restricted to linear systems is related to Gaussian Elimination [49].

1. Use one of the equations to express one of the unknowns as a function of the rest. This can be done for linear equations by using zero-crossings, thus only unknowns with non-zero coefficients can be eliminated from a linear equation.

2. Eliminate the chosen unknown from the rest of the equations by replacing it with the expression obtained in the first step.

3. Recursively solve the system of rewritten equations until the substitution expression has no unknowns (i.e. it is a constant value). This expression gives us the value of the last unknown.

4. Once the residual system is solved, compute the value of the eliminated unknown by using the expression determined in the first step and the values of the unknowns found in the recursive step.

## 3.2 Substitution as an Operation on Functions

To write an equation solver based on this method, we could use a symbolic representation and implement symbolic substitution, renaming, simplification, etc.

Alternatively, we can observe that the elimination, substitution, and rewriting described above are easily expressed in terms of functional abstraction and composition: If we somehow represent our equations as functions of the unknowns, to substitute one of the unknowns by a function of the others, we compose the original equation with the function expressing the unknown. For example, using Scheme notation, if an equation is represented by some function of three arguments, in some not-yet-specified way,

```
Eqn = (lambda (x y z)
        ... x ... y ... z ...)
```

and our substitution is

```
Sbx = (lambda (y z)
        (/ (- (+ y 5) z) 2))
```

the elimination step consists of replacing the original equation with the new equation

```
Eqn' = (lambda (y z)
         ... (/ (- (+ y 5) z) 2) ... y ... z ...)
```

but this is just (in general) the function

```
Eqn' = (lambda (y z)
         (Eqn (Sbx y z) y z))
```

To change to a function-based view, we can model each equation as a function of the unknowns whose zeros we want to find. A zero of a function is a vector of argument values that the function maps to zero. A system of equations is then just a set of functions that we want to zero simultaneously. That is, to solve a system of equations represented as a set of functions is to find a common vector of arguments that result in zero when supplied to each of the functions. In our example, the functions might be

```
F1 = (lambda (x y z) (- (+ (* 2 x) (- y) z) 5))

F2 = (lambda (x y z) (- (+ x y z) 6))

F3 = (lambda (x y z) (- (+ x (- y) z) 2))
```

To illustrate the substitution method in the function-based model, we can solve for `x` in terms of `y` and `z` by constructing the following functions, and then use `Sx` on the solution of the system formed by `F2'` and `F3'` to find the value of `x`.

```
Nx = (lambda (y z) (F1 0 y z))

Dx = (lambda (y z) (- (F1 1 y z) (Nx y z)))

Sx = (lambda (y z) (/ (- (Nx y z)) (Dx y z)))

F2'= (lambda (y z) (F2 (Sx y z) y z))

F3'= (lambda (y z) (F3 (Sx y z) y z))
```

These functions may seem to have been pulled out of thin air, but they are not hard to understand if we consider the case of a simple line

```
F  = (lambda (x) (+ (* D x) N))
```

corresponding to the equation

$$Dx + N = 0$$

Clearly, the solution to this equation is $S = -N/D$, but in the function model we do not have direct access to $N$ and $D$, although they can be computed easily:

```
N = (F 0)
D = (- (F 1) N)
```

Compare `N` and `D` with `Nx` and `Dx` above, and note that for fixed values of `y` and `z`, `F1` is a simple line with the single variable `x`. `Sx` is then the function that computes the value of `x` that zeros the first equation given arbitrary values for the remaining unknowns. This is precisely the substitution function that we need to reduce the system, and the expressions for `F2'` and `F3'` should now be clear.

To make the method even more concrete, we can expand out and simplify the functions `Nx`, `Dx`, `Sx`, `F2'`, and `F3'`:

```
Nx = (lambda (y z) (F1 0 y z))
   = (lambda (y z) (- (+ (- y) z) 5))
   = (lambda (y z) (- z (+ y 5)))

Dx = (lambda (y z) (- (F1 1 y z) (Nx y z)))
   = (lambda (y z) (- (- (+ 2 (- y) z) 5) (- z (+ y 5))))
   = (lambda (y z) (- (+ 2 (- z (+ y 5))) (- z (+ y 5))))
   = (lambda (y z) 2)

Sx = (lambda (y z) (/ (- (Nx y z)) (Dx y z)))
   = (lambda (y z) (/ (- (- z (+ y 5))) 2))
   = (lambda (y z) (/ (- (+ y 5) z) 2))

F2'= (lambda (y z) (F2 (Sx y z) y z))
   = (lambda (y z) (- (+ (/ (- (+ y 5) z) 2) y z) 6))
   = (lambda (y z) (/ (- (+ (- (+ y 5) z) (* 2 y) (* 2 z)) 12) 2))
   = (lambda (y z) (/ (- (+ (* 3 y) z) 7) 2))

F3'= (lambda (y z) (F3 (Sx y z) y z))
   = (lambda (y z) (- (+ (/ (- (+ y 5) z) 2) (- y) z) 2))
   = (lambda (y z) (/ (- (+ (- (+ y 5) z) (* -2 y) (* 2 z)) 4) 2))
   = (lambda (y z) (/ (- (+ (- y) z) -1) 2))
```

where the expansion and simplification are carried out only to illustrate that `F2'` and `F3'` correspond to equations 3.5 and 3.6 directly.

By representing equations as functions, and in turn, functions as procedures, we have found a way to compute substitutions without manipulating expressions, but instead by appropriately abstracting and composing the functions (procedures) that represent the equations.

In essence, we have outlined the basic structure of an elegant linear equation solver, and we can envision writing it, without making any reference to algebraic simplification, variable renaming, or substitutions within expressions.[2] These operations may be desirable to obtain simple answers, and to ensure that cancellation does not produce spurious divisions by zero, or incorrect results due to round-off error, overflow, or underflow, but they are not inherently required to solve the equations. These symbolic operations are necessary only as a consequence of choosing expressions as the representation of our equations, just like scaling of points, segments, and arcs is not inherent in the picture language, but only a consequence of choosing lists of segments to represent our pictures. Functions (procedures) capture the behavior of our equations, and we can eliminate and substitute variables naturally by composing and abstracting.

## 3.3 Additional Concerns to the Application of the Method

The method as outlined so far is, substantively, a numerical method, and will work for systems containing arbitrary functions as long as we can find functions linear (actually affine) on some of the unknowns. Because our method eliminates each unknown using only one function at a time, it only depends on the function being affine with respect

[2]The idea of using procedures in this way, and the overall structure of the solver, are due to Harold Abelson and Gerald J. Sussman.

to the single parameter being eliminated. The method can consequently reduce some more general non-linear systems.

For example, if in the system of equations 3.1- 3.3 we perform the following substitutions,

$$x = uv \tag{3.9}$$
$$y = vw \tag{3.10}$$
$$z = wu \tag{3.11}$$

we arrive at the system

$$\begin{aligned} 2uv - vw + wu &= 5 \\ uv + vw + wu &= 6 \\ uv - vw + wu &= 2 \end{aligned}$$

in which no equation is linear, but each equation is affine in each parameter individually. The method outlined above reduces this system to an easily factorable quintic equation.[3]

Furthermore, not all of the functions need to be algebraic. The system will eliminate unknowns by finding functions affine in some unknowns, and express the solution in terms of the solution of the reduced system, which can then be solved by iterative methods.

Before we examine a solver written using this method, we need to consider the possibility that our system of equations may be over- or under-constrained. A system of equations is over-constrained when it has no solutions, and under-constrained when there are an infinite number of solutions. Over-constrained systems can be handled easily by returning some pre-established object that is otherwise an invalid solution. Under-constrained systems are a little more complicated, but it is not difficult to arrive at a useful convention by examining linear systems. A system of linear equations is underconstrained only if it has fewer equations than unknowns, after removing algebraically dependent equations. If we follow our substitution and elimination method

[3]Quintics are not always factorable [5, 35], but this one has no constant coefficient, and the quartic factor is bi-quadratic.

on such a system, we will reach a point when we have expressed some of the unknowns in terms of the rest, but there are no equations left that can be used find the values of the remaining unknowns. At this point, *any* set of values for the remaining unknowns leads to a solution of the initial system. The solutions of the initial system form a subspace whose dimensionality is the number of unknowns remaining when we reach this point.

To handle this situation, we can view a solution to a system of equations on $n$ unknowns, not as a single $n$-dimensional point, but as a function mapping some number $m$ of parameters to $n$ values of the unknowns that zero the system. `m` is the dimensionality of the solution subspace. If the system has a unique solution, $m$ will be zero and the solution function will be a constant function, while for a system whose solutions all lie on a plane, $m$ will be two, and the solution function will map arbitrary points in $\mathcal{R}^2$ (or $\mathcal{C}^2$) to points in the plane of solutions.[4]

For example, a solution function for the original example system would be

```
(lambda ()
  (tuple 3 2 1))
```

while a solution function for the under-constrained system of equations

$$\begin{aligned} 3x + 2y &= 1 \\ 3y + 2z &= 1 \\ 9x - 4z &= 1 \end{aligned}$$

might be

```
(lambda (z)
  (tuple (/ (+ 1 (* 4 z)) 9)
         (/ (- 1 (* 2 z)) 3)
         z))
```

where all solutions lie on a line.

[4]This method of returning a function of some continuous parameter space can be extended to return a function and a domain to encompass multiple roots. Some of the components of the domain would be continuous, while others would be discrete. The function would use continuous arguments as described here, and the discrete arguments to choose roots of unity.

## 3.4 The Solver

Let us now examine a solver written using these ideas. Our solver takes a list of procedures representing the equations, and returns two values, a residual system represented as a list of procedures, and a solution procedure to invoke on solutions of the residual system to produce solutions of the original system. If the residual system is empty, the original system is completely solved, and the number of parameters of the solution procedure is the dimensionality of the solution space. If it is not empty, the solver could not solve the system fully, and some other method will have to be used on the residual system.

Our top-level procedure performs some consistency checking and prepares the null solution, i.e., the composition identity of $N$ parameters. The real work is done by `solve-aux` and `try-methods`.

```
(define (solve S)
  (if (null? S)
      (error "solve: No equations")
      (let ((arity (proc/arity (car S))))
        (if (there-exists? (cdr S)
               (lambda (eqn)
                 (not (= (proc/arity eqn) arity))))
             (error "solve: Inconsistent number of unknowns" S)
             (solve-aux S (composition-identity arity))))))

(define (solve-aux S F)
  (if (null? S)
      (values S F)
      (try-methods *solution-methods*
                   (map simplify-eqn S)
                   F)))

(define (try-methods methods S F)
  (if (null? methods)
      (values S F)
      (with-values (lambda () ((car methods) S))
        (lambda (S1 f1)
          (if (and (= (length S1) (length S))
                   (= (proc/arity (car S))
                      (proc/arity (car S1))))
```

```
                ;; Method failed
                (try-methods (cdr methods) S F)
                ;; Method eliminated a variable or an equation
                (solve-aux S1 (compose F f1)))))))
```

`solve-aux` takes two arguments: the residual system that we still need to solve, and a partial solution, i.e., a function that maps solutions of the residual system into solutions of the original system. If there are no equations left to solve, we are done, and the partial solution is the total solution. Otherwise `solve-aux` uses `try-methods` to try all the known solution methods.

A solution method is a procedure that takes a system of equations and returns a residual system and a partial solution. As before, a partial solution maps solutions to the residual system into solutions of the system given to the solution method. Solution methods typically remove some equations or some unknowns from the original system. When unknowns are eliminated, the partial solution computes the eliminated unknowns in terms of the remaining unknowns. When the method eliminates equations without eliminating unknowns, perhaps because the equations have become tautologies (identically zero) after substitution, the partial solution is the identity function on the vector of unknowns.

`try-methods` tests whether the first method in the list reduces the system, and if not, it tries the remaining methods. When a method succeedes, `try-methods` calls `solve-aux` on the residual system, and the corresponding overall partial solution that is just the composition of the previous overall partial solution and the local partial solution that the method returned.

`simplify-eqn`, used in `solve-aux`, can be used to remove denominators from rational functions and to perform other similar tasks. For our solver we can define it as follows:

```
(define (simplify-eqn f)
  (if (ratfun? f)
      (ratfun/numerator f)
      f))
```

Our list of methods can contain any procedures that can reduce the system. We can use the following list:

```
(define *solution-methods*
  (list (lambda (S)
          (affine-eliminate S))
        (lambda (S)
          (quadratic-eliminate S))
        (lambda (S)
          (constant-eliminate S))
        (lambda (S)
          (iterative-solve S))))
```

`affine-eliminate` embodies the method outlined earlier and is examined below. `quadratic-eliminate` is similar, but solves quadratic equations.[5] `constant-eliminate` removes tautologies, i.e. functions that are identically zero. These equations can appear after reducing a system with algebraically dependent equations. `iterative-solve` is a trivial root finder that can be used to solve uni-dimensional equations not amenable to other methods.

`affine-eliminate` is defined as follows.

```
(define affine-eliminate
  (equation-seeker (lambda (f i)
                     (and (affine? f i)
                          (not (independent? f i))))
                   (lambda (f i ignore)
                     (affine-invert f i))))
```

`equation-seeker` constructs a solution method that the solver can use. It takes a predicate procedure and an *inversion* procedure. The predicate procedure tests whether an unknown can be eliminated by using an equation. It is given the equation and the index of the unknown as arguments. The inversion procedure eliminates the corresponding unknown by expressing it in terms of the remaining unknowns. `equation-seeker` tries all the equations and all the argument indices:

[5] Quadratic equations typically have two solutions. Without extending our solutions to encompass discrete parameters, `quadratic-eliminate` can produce both by backtracking. The backtracking mechanism is not germane to the discussion.

```
(define (equation-seeker predicate inverter)
  (lambda (S)
    (let ((n-unknowns (proc/arity (car S))))
      (define (find-eqn unk-index)
        (let find-eqn ((S S) (eqns '()))
          (cond ((null? S)
                 ;; No more equations?
                 ;; -> Try the next unknown
                 (try (1+ unk-index)))
                ((predicate (car S) unk-index)
                 ;; Can this equation be inverted to eliminate
                 ;; the M-th unknown?
                 => (lambda (pred-result)
                      (let ((inversion
                             (inverter (car S) unk-index pred-result))
                            (n* (-1+ n-unknowns)))
                        ;; Reduce the system by eliminating the
                        ;; equation and the unknown,
                        ;; and produce a partial solution function
                        ;; that computes the value of the eliminated
                        ;; unknown by using the rest, and inserts it
                        ;; at the right position in the argument
                        ;; vector.
                        (values
                         (map (lambda (f)
                                (introduce f inversion unk-index))
                              (append (reverse eqns) (cdr S)))
                         (aggregate* (*segment n* 0 unk-index)
                                     inversion
                                     (*segment n* unk-index n*))))))
                (else
                 (find-eqn (cdr S) (cons (car S) eqns))))))

      (define (try unk-index)
        (if (< unk-index n-unknowns)
            (find-eqn unk-index)
            ;; No unknowns left, return the original system
            ;; and a dummy partial solution
            (values S (composition-identity n))))

      ;; Try the first unknown
      (try 0))))
```

`equation-seeker` does not choose good unknowns to eliminate. It chooses the first unknown that satisfies the predicate, and this is rarely the best choice. If a system of linear equations is triangular, it may not choose the unknown that appears by itself in an equation. Alternatively, if the system is dense, the best choice is often the unknown with the largest coefficient, but this is unlikely to be the first one found. At any rate, better choices can be implemented – this version is illustrative, not exemplary.

The code for `equation-seeker` is elaborate, but mostly straightforward. It consists of two nested loops. The outer iterates over all the indices of the unknowns. The inner iterates over all of the equations for each index. `equation-seeker` uses the predicate on pairs of equations and indices, until the predicate succeeds, and then it uses the inversion function to compute the substitution, and then, the residual system and the partial solution.

The only complication is the construction of the residual system and the partial solution when the method succeeds. The inversion function computes the value of the eliminated unknown in terms of the rest, but the partial solution must take the values of the remaining unknowns and produce a solution vector with them and the eliminated unknown in the correct positions. To construct the partial solution, `equation-seeker` uses `aggregate*` and `*segment`, the argument-list analogs of the list operations `append` and `subseq` [57].

The other equations are rewritten by substituting the eliminated unknown for its value in terms of the remaining unknowns. This task is performed by the procedure `introduce`, which takes two procedures, $f$ and $g$, and one argument index, $n$. $G$ must take one less argument than $f$. `introduce` returns a procedure, $h$, that when invoked, passes its arguments to $g$, collects its result, inserts it at the $n$-th position in the arguments passed to $h$, and passes the resulting vector of arguments to $f$, returning its result. E.g.,

```
(introduce (lambda (x y z) (+ (* x y) z))
           (lambda (x z) (- x z))
           1)

≡ (lambda (x z)
```

```
    (+ (* x (- x z)) z))
```

`affine-invert`, used in the definition of `affine-eliminate`, embodies the substitution method that we outlined earlier.

```
(define (affine-invert f i)
  ;; f(xi) = D * xi + N            =>      N = f(0)
  ;;                                       D = (f(1) - N)
  ;; For a zero,                  xi = ((f(xi) - N) / D) = ((-N) / D)
  (let* ((N (fix-argument/drop-parameter f i 0))
         (D (combine -
                     (fix-argument/drop-parameter f i 1)
                     N)))
    (combine /
             (compose negate N)
             D)))
```

`fix-argument/drop-parameter` takes a procedure, an argument index $i$, and a constant value, and returns the procedure that takes one less argument and corresponds to the original with the $i$-th argument substituted by the constant. For example,

```
(fix-argument/drop-parameter (lambda (x y)
                                (- (* x y) (+ x 7)))
                              1
                              29)

≡ (lambda (x)
    (- (* x 29) (+ x 7)))
```

`combine` produces a procedure $p$, that, when invoked, calls all but the first argument to `combine` on the arguments to $p$, collects all of the results, and invokes the first argument to `combine` on all the collected results, returning what this procedure returns. For example,

```
(combine -
         (lambda (a b c) (* a (+ b c)))
         (lambda (x y z) (+ (* z x) (* z y))))

≡ (lambda (e f g)
    (- (* e (+ f g))
       (+ (* g e) (* g f))))
```

## 3.5 The Limitations of Opaque Procedures

Besides procedural utilities such as `combine`, `fix-argument/drop-parameter`, `introduce`, etc., we have yet to write `affine?`, `ratfun?`, and `ratfun/numerator` to finish implementing our solver.

A function is affine in some parameter, if, for any fixed given values of the remaining parameters, the function is a line. In other words, function $\mathcal{F}$ is affine in its first argument (similarly for other arguments) if we can write it as

$$\mathcal{F}(x_1, x_2, \ldots, x_m) \quad = \quad \mathcal{F}_1(x_2, \ldots, x_m) * x_1 + \mathcal{F}_0(x_2, \ldots, x_m)$$

A sufficient condition, when $\mathcal{F}$ is partially differentiable twice with respect to its first argument, is that this partial derivative is identically zero. That is, $\mathcal{F}$ is affine in its first argument (similarly for other arguments) if the following holds:

$$\frac{\partial^2 \mathcal{F}}{\partial x_1^2} = 0$$

If our procedures are opaque, we cannot easily differentiate exactly, just numerically.[6] We can try another approach. If $\mathcal{F}$ is affine in its first argument, then

$$\begin{aligned} \mathcal{H}(x_1, x_2, \ldots, x_m) &= \mathcal{F}(x_1, x_2, \ldots, x_m) - \mathcal{F}(0, x_2, \ldots, x_m) \\ &= \mathcal{F}_1(x_2, \ldots, x_m) * x_1 \end{aligned}$$

must be linear in its first argument. This immediately leads to the following definition of `affine?`

```
(define (affine? f n)
  (linear? (combine - f (fix-argument/keep-parameter f n 0))
           n))
```

where `fix-argument/keep-parameter` is similar to `fix-argument/drop-parameter` (used earlier), but produces a result of the same arity as its first argument.

[6]As observed by Gerald J. Sussman and Dan Zuras, exact differentiation can be carried out by using non-standard analysis [34]. This presumes that our arithmetic primitives are extended to accomodate infinitesimals.

How do we test for linearity? Rejecting differentiation again, we can make direct use of the definition of linearity. A function $\mathcal{H}$ is linear in its first argument (similarly for other arguments) if the following identities hold for all values of the $x_i$, $k$, and $y$.

$$\begin{aligned}
\mathcal{H}(k * x_1, x_2, \ldots, x_m) &= k * \mathcal{H}(x_1, x_2, \ldots, x_m) \\
\mathcal{H}(x_1 + y, x_2, \ldots, x_m) &= \mathcal{H}(x_1, x_2, \ldots, x_m) + \mathcal{H}(y, x_2, \ldots, x_m)
\end{aligned}$$

This is equivalent to requiring that the functions $\mathcal{P}$ and $\mathcal{S}$ (for product and sum, respectively), defined as follows, be identically zero.

$$\begin{aligned}
\mathcal{P}(x_1, x_2, \ldots, x_m, k) &= \mathcal{H}(k * x_1, x_2, \ldots, x_m) - k * \mathcal{H}(x_1, x_2, \ldots, x_m) \\
\mathcal{S}(x_1, x_2, \ldots, x_m, y) &= \mathcal{H}(x_1 + y, x_2, \ldots, x_m) - (\mathcal{H}(x_1, x_2, \ldots, x_m) + \mathcal{H}(y, x_2, \ldots, x_m))
\end{aligned}$$

We can program this test as follows:

```
(define (linear? f n)
  (let* ((m (proc/arity f))
         (m* (1+ m)))
    (let* ((f-of-nth
            (lambda (substitution)
              (combine f
                       (segment m* 0 n)
                       substitution
                       (segment m* (1+ n) m))))
           (f-of-nth-and-mth
            (lambda (combiner)
              (f-of-nth (combine combiner
                                 (project m* n)
                                 (project m* m))))))
      (let ((scale-test
             (combine -
                      (f-of-nth-and-mth *)
                      (combine *
                               (project m* m)
                               (compose f (segment m* 0 m)))))
            (sum-test
             (combine -
                      (f-of-nth-and-mth +)
                      (combine +
                               (compose f (segment m* 0 m))
                               (f-of-nth (project m* m))))))

        (and (identically-zero? scale-test)
             (identically-zero? sum-test))))))
```

We now need to implement `identically-zero?`, and of course, we cannot. `identically-zero?` is undecidable in general, but that is not the fundamental issue here, since suitable restrictions are decidable, and we can let the solver use other methods when `identically-zero?` fails because it is conservative.

The problem is that if procedures are opaque, we can only invoke them, not *look inside*. The best we can do is to invoke the procedure on a finite number of $n$-dimensional points [42]. If the result at any of them is not zero, `identically-zero?` can truthfully return false, however, even if the function returns zero at every point tested, there is no guarantee that the result is identically zero. Although choosing random points makes for a very reliable and fast test, ultimately we can use procedure invocation only to reject, never to accept, except heuristically.

Of course, `ratfun?`, etc. also lead us to the same problem. When procedures are completely opaque, they cannot be used to represent functions that we want to compare to other functions, unless their domains are finite, and well known, even if the comparison is only approximate to avoid undecidability.

As in the Henderson language scenario, we can resort to unsavory tricks such as *switching* the meaning of the arithmetic operators to handle symbolic quantities, and then invoke our functions on symbolic arguments and examine the result, but there is no guarantee that all the functions are written in terms of the operators that we have switched.

Similarly, we can change the interface to our functions, and change `combine`, etc., accordingly, but this will obscure our basic representation decision, force us to maintain dual representations, and demand that users of the solver write their functions using this interface if they want to pass them to the solver as equations.

## 3.6 Overcoming Opacity

As in the Henderson language example, we can require that our language allow us to inspect the structure of procedures. Even before this point, we have already assumed a limited form of inspection, since `proc/arity`, used in `solve`, `try-methods`, `equation-seeker`, and `linear?`, cannot be implemented if procedures are completely opaque.

In our extended Scheme, we can use `tlambda` when writing our equations, and write our procedure utilities appropriately, so that all the generated procedures are translucent when the inputs are.

We can now write a conservative `identically-zero?` as follows:

```
(define (identically-zero? f)
  (and (ratfun? f)
       (match? (constant (proc/arity f) 0)
               (ratfun/simplify f))))
```

`ratfun?` can be written as follows:

```
(define (ratfun? proc)
  (and (tproc? proc)
       (%ratfun? proc)))

(define (%ratfun? proc)
  (let* ((m (tproc/arity proc))
         (mc (constant m #?CONST))
         (proj (let make ((i 0))
                 (if (>= i m)
                     '()
                     (cons (project m i)
                           (make (+ i 1))))))
         (ops (map (let ((mp1 (restrict #?P1 m))
                         (mp2 (restrict #?P2 m)))
                     (lambda (op)
                       (combine op mp1 mp2)))
                   (list + - * /))))

    (let test ((proc proc))
      (let find-operator ((ops ops))
        (cond ((null? ops)
```

```
           (cond ((match? mc proc)
                  => (lambda (match)
                       (number? (match/lookup match #?CONST))))
                 (else
                  (let find-projection ((proj proj))
                    (and (not (null? proj))
                         (or (match? (car proj) proc)
                             (find-projection (cdr proj)))))))))
          ((match? (car ops) proc)
           => (lambda (match)
                (and (test (match/lookup match #?P1))
                     (test (match/lookup match #?P2)))))
          (else
           (find-operator (cdr ops)))))))
```

`%ratfun?` uses the matcher to check whether its argument is: a sum, product, etc.; a constant function; or a projection function. If the argument is a sum, product, etc., it tests the operands recursively.

`ratfun/simplify` and `ratfun/numerator` can be written directly, like `ratfun?`, or, alternatively, can be written by translating their argument to an alternate representation, simplifying the result, and translating back. The alternate representation may be any ordinary representation for rational functions, such as a pair of polynomials with sparse coefficients.

`ratfun->rf`, shown in fig. 3-1, translates a translucent procedure that implements a rational function into an alternate representation, manipulated abstractly by the operations `constant->rf`, `projection-rf`, `rf+`, `rf-`, `rf*`, and `rf/`.

## 3.7 Loose Ends

The solver described above is mostly complete, but a few pieces are still missing. For example, `affine-eliminate` depends not only on `affine?` but also on `independent?`, which can be coded as follows,

```
(define (independent? f param)
  (identically-zero?
    (combine -
```

```
(define (ratfun->rf proc)
  (let ((m (procedure/arity proc)))
    (let ((mc (constant m #?CONST))
          (mp1 (restrict #?P1 m))
          (mp2 (restrict #?P2 m))
          (proj (let make ((i 0))
                  (if (>= i m)
                      '()
                      (cons (project m i)
                            (make (+ i 1))))))
          (bases (let make ((i 0))
                   (if (>= i m)
                       '()
                       (cons (projection-rf m i)
                             (make (+ i 1))))))
          (number-ops (list + - * /))
          (rf-ops (list rf+ rf- rf* rf/)))

      (let test ((proc proc))
        (cond ((match? mc proc)
               ;; Is it a constant function?
               => (lambda (match)
                    (constant->rf m (match/lookup match #?CONST))))
              (else
               (or
                ;; Is it a projection function?
                (let find-projection ((proj proj) (bas bas))
                  (and (not (null? proj))
                       (if (match? (car proj) proc)
                           (car bas)
                           (find-projection (cdr proj)
                                            (cdr bas)))))
                ;; Is it a sum, product, etc?
                (let find-op ((ops number-ops)
                              (rf-ops rf-ops))
                  (and (not (null? ops))
                       (let ((match
                              (match? (combine (car ops) mp1 mp2)
                                      proc)))
                         (if match
                             ((car rfops) (match/lookup match #?P1)
                                          (match/lookup match #?P2))
                             (find-op (cdr ops) (cdr rf-ops))))))
                (error "ratfun->rf: Not a ratfun"
                       proc))))))))
```

Figure 3-1: Conversion from Procedures to Other Representations

```
                 f
                 (fix-argument/keep-parameter f param 0))))
```

which tests whether $\mathcal{F}$ is independent from $x_1$ (or any other parameter) by testing whether $\mathcal{F}'$, defined as follows, is identically zero.

$$\mathcal{F}'(x_1, x_2, \ldots, xn) \quad = \quad \mathcal{F}(x_1, x_2, \ldots, xn) - \mathcal{F}(0, x_2, \ldots, xn)$$

The missing procedures are straightforward, or similar to those already described, but some care must be exercised when writing utilities such as `combine` and `fix-argument/drop-parameter`.

Our solver depends in several places on the ability to determine the arity of the procedures that represent equations. It uses this ability to find the number of unknowns remaining, and to decide when a solution method has eliminated some of them.

Procedures such as `combine`, etc., might appear to be easily expressible in ordinary Scheme with definitions such as

```
(define (combine combiner . elements)
  (lambda args
    (apply combiner
           (map (lambda (element)
                  (apply element args))
                elements))))
```

but even assuming that the result was a translucent procedure, it would have lost all arity information, and the solver would no longer be able to find the number of unknowns remaining. Writing our utilities in this way would force us to maintain and pass around the arity, rather than extract it from the procedures.

The definitions of these utilities can preserve this information if they are written in a manner analogous to the version of `compose` in chapter 2. For example, we can write `combine` as follows:

```
(define (combine collect . fs)
  (compose collect (apply aggregate* fs)))
```

Then, we can define `aggregate` and `aggregate*` as follows:

```
(define (aggregate* f1 . fs)
  (let loop ((f1 f1) (fs fs))
    (if (null? fs)
        f1
        (loop (aggregate f1 (car fs))
              (cdr fs)))))

(define (aggregate f g)
  ;; h = (aggregate f g)
  ;; => (h . args) = (concat (f . args) (g . args))
  (let ((ftakes (proc/arity f))
        (freturns (proc/nvalues f))
        (gtakes (proc/arity g))
        (greturns (proc/nvalues g)))
    (if (not (= ftakes gtakes))
        (error "aggregate: Incompatible" f g)
        (let ((names (make-list-of-names (+ ftakes freturns greturns)
                                         '(f g))))
          (let ((formals (list-head names ftakes))
                (rest (list-tail names ftakes)))
            (let ((fvals (list-head rest freturns))
                  (gvals (list-tail rest freturns)))
              ((tproc/make*
                `(lambda (f g)
                   (lambda ,formals
                     (tlet (,fvals (f ,@formals))
                       (tlet (,gvals (g ,@formals))
                         (tuple ,@fvals ,@gvals))))))
               f g)))))))
```

aggregate* was used in equation-seeker in conjunction with *segment, with the latter defined as follows:

```
(define (*segment takes low high)
  (if (not (<= 0 low high takes))
      (segment takes 0 0)
      (segment takes low high)))

(define (segment takes low high)
  (if (not (<= 0 low high takes))
      (error "segment: Invalid range" takes low high)
      (let* ((formals (make-list-of-names takes '()))
             (keep (sublist formals low high)))
        (tproc/make* `(lambda ,formals
                        (tuple ,@keep))))))
```

fix-argument/drop-parameter can be written in terms of these procedures as follows:

```
(define (fix-argument/drop-parameter f n konst)
  ;; Fix argument n of f to constant konst.
  ;; The resulting procedure takes one less argument.
  #|
  ;; Equivalent to
  (introduce f
             (constant (-1+ (tproc/arity f))
                       konst)
             n)
  |#
  (let ((m* (-1+ (tproc/arity f))))
    (combine f
             (*segment m* 0 n)
             (constant m* konst)
             (*segment m* n m*))))
```

The remaining utilities (constant, fix-argument/keep-parameter, introduce, etc.) can easily be written in a similar style.

## 3.8 Summary

We examine the construction of an equation solver where equations are represented as procedures. The power of this choice is that it allows us to compute and express substitutions and solutions without reference to expressions, variables, or renaming. This independence from an expression representation allows us, among other things, to mix algebraic with iterative methods within the same framework.

Ordinary procedures are unsuitable because they cannot easily support recognition and discrimination, abilities necessary to examine our equations in order to choose an appropriate solution method. Translucent procedures, which can be composed and abstracted as easily as ordinary procedures, also permit limited inspection, enabling us not only to construct a solution by composition, but also to choose solution methods by examining the equations.

# Chapter 4

# Metacircular Interpreters and Compilers

Interpreter construction is a scenario where a procedural representation is natural, elegant, and efficient. As in the previously examined scenarios, however, opaque procedures cannot be used as the basic representation because they preclude the implementation of certain operations on our abstract representation. We will see that translucent procedures overcome this problem nicely.

An expression to be evaluated can be abstracted as a behavior modulated by an environment. In this chapter we will see how this observation leads naturally to a procedural representation for expressions in our interpreter. However, this representation requires translucent procedures in order to make interpreter states invertible, hence debuggable.

## 4.1 Why Use Interpreters?

Interpreter construction is an important topic in the implementation of interactive languages (e.g. APL [27], Basic, Forth, Lisp [57], Scheme [11, 1], Smalltalk [14], csh [4]).

Interpreters are used in these languages because a high priority in the implementation of such languages is to keep the time of the edit-run cycle as low as possible. The edit-run cycle is usually more time consuming when a native-code compiler, even a non-optimizing compiler, is inserted in the cycle. In addition, interpreters are often preferred for debugging for a variety of reasons:

- The reduced time of the edit-run cycle allows the programmer to explore more possibilities in the same amount of time. The programmer can more easily write and run test code, or add debugging output to the code.

- Interpreters often have a more direct execution model, not subject to optimizations that may rearrange or eliminate code or variables.

- Debuggers can more easily invert and display the state of the computation because interpreters have a known finite number of states.

There is a natural tension in the design and implementation of an interpreter between its speed and its simplicity. As always, simplicity leads to increased maintainability. Simple interpreters are easy to write, but often perform poorly. Complicated interpreters perform well, but are harder to debug and maintain, and sometimes make debugging the interpreted code more difficult.

The most common design technique for interactive languages is to use a hybrid of compilation and interpretation. The source language is compiled to a simple binary code, often a linear byte code for a stack architecture, whose interpretation can be carried out easily and efficiently by a straightforward program. The byte code is designed for fast execution and for straightforward translation, so the compilation step, unlike compilation to native code, is quick.

The efficiency of byte-coded interpreters is often adequate, but the *compilation* step usually makes debugging interpreted code harder. There are two common solutions to this problem: we can make the output of the byte code compiler invertible, or, alternatively, we can use techniques similar to those used by full-fledged native-code compilers, i.e. the compiler can output of additional data structures mapping code addresses to debugging information. The first solution impacts negatively on performance—it reduces the range of possible optimizations; the second implies too large an implementation effort for "lean-and-mean" interpreters.

A second common technique used in interpreter design, particularly for the Forth language, is to use threaded interpreters [41]. A threaded interpreter is another hybrid technique. The source language is translated locally into pre-packaged template machine code sequences with slots for the translated subexpressions. The code operates by executing the templates, which decide when and how to invoke the templates contained in the slots corresponding to the subexpressions and to combine their results.

Threaded interpreters are often very fast, but must be ported individually to every new architecture and have the same debugging problems as byte-coded interpreters.

Threaded interpreters are typically faster than byte-coded interpreters because they eliminate one level of decoding. Byte-coded interpreters translate the source language into a small linear instruction set that must be decoded by software at run-time. The compiler to threaded code produces a graph of native code segments that is executed directly by the hardware, with no further software decoding. Although the code in threaded interpreters is often hard on processor pre-fetch units, because of all the jumping around, byte-coded interpreters often use large dispatch tables to decode the op-codes, and short code sequences in each entry, imposing a similar burden on the pre-fetch unit.

Threaded interpreters, besides being hard to port, are also harder to write. The interpreter implementor must write and tune the code sequences in assembly language. However he must also be very careful about maintaining the correct state, and think carefully about how the standard code sequences combine at runtime in terms of registers and stacks, and other low-level structures.

## 4.2 Automatically Generated Threaded Interpreters

The typical implementation of some languages (Lisp, Scheme) contains both an interpreter and a native-code compiler. The interpreter is used for interaction, and to reduce the time of the edit-run cycle; the compiler is used on previously-tested modules that should run quickly, at some expense in debugging ability.

Even though the compiler is available to accelerate code, the speed of the interpreter should not be neglected. If it is very slow, programmers will only use the compiler. It must also be no more difficult to debug interpreted code than compiled code, since that is an important reason for its existence.

Fortunately, there is a simple technique that can be used to implement threaded interpreters for such systems. The technique makes use of the effort in porting the native-code compiler, and allows the interpreter to be written entirely in the source language, using the native-code compiler to gain its efficiency.

This technique, introduced by Feeley and Lapalme [21] primarily for code generation (rather than direct execution) consists of translating expressions into closures (procedures). An expression is translated into a procedure that accepts a single argument, namely a representation of the runtime environment where the expression is to be evaluated. The procedure invokes the procedures corresponding to the subexpressions on suitable environments, and combines the results, or chooses among alternatives according to these results.

The beauty of this technique is that it abstracts the meaning of an expression into a behavior, implemented concretely by the procedure that represents the expression. For execution, the only important aspect of an expression is what actions and values it manipulates when evaluated in an environment; these actions and value manipulations modularized by the runtime environment, are what the procedure representing the expression captures exactly.

To illustrate the technique, we can explore the implementation of a Scheme interpreter, using Scheme as the implementation language as well. Such an interpreter is metacircular, but given that in reality it *compiles* the source language into data structures (albeit executable), it should more properly be called a metacircular compiler.

For expository reasons, the code that appears below is written in a concrete style, with little data abstraction, and with no consistency checks. A *production* version would be more abstract, but this abstraction would not only lengthen the code that appears below, but would also obscure its workings. Note also that a full metacircular compiler is too long to include here, but the following code segments should suggest how the rest of the code works.

The top-level of the compiler is a simple syntactic dispatch:

```
(define (execute expr rtenv)
  ((compile expr) rtenv))

(define (compile expr)
  (compile-expr expr (ct/initial)))

(define (compile-expr expr ctenv)
  (cond ((pair? expr)
         (case (car expr)
           ((LAMBDA)
            (compile-lambda expr ctenv))
           ((IF)
            (compile-if expr ctenv))
           ...
           (else
            (compile-combination expr ctenv))))
        ((symbol? expr)
         (compile-variable expr ctenv))
        (else
         (compile-constant expr ctenv))))
```

`ctenv` is a compile-time model of the runtime environment. Except for the top-level (or global) runtime environment, runtime environments grow by adding frames created when interpreted procedures are invoked. A new frame binds the procedure's formal parameters to the actual arguments passed to the procedure. Runtime environments are captured into runtime closures resulting from the execution of `lambda` expressions. The compile-time environment models the layout of the runtime environment that will be manipulated by the translated code.

Some examples of the code generators:

```
(define (compile-constant const ctenv)
  ctenv                                   ; ignored
  (LAMBDA (rtenv)
    ;; rtenv                              ; ignored
    const))
```

The value of a constant does not depend on the runtime environment. The code generated for a constant ignores the environment and returns the value of the constant.

```
(define (compile-variable var ctenv)
  (ct/lookup ctenv var
             (lambda (depth offset)
               (if (not depth)
                   (compile-global-variable var ctenv)
                   (compile-lexical-lookup depth offset)))))

(define (compile-lexical-lookup depth offset)
  (LAMBDA (rtenv)
    (vector-ref (list-ref rtenv depth) offset)))
```

To compile a variable, we examine the compile-time environment. If the variable is present—i.e. it was introduced by a `lambda` expression in the current compilation unit—the path in the compile-time environment determines the path in the runtime environment; we generate code that extracts the value from the runtime environment by following that path. The representation of runtime environments (as lists of vectors) is fixed by the compiler, which generates procedures that directly access their components.

If the variable is not found in the compile-time environment, it is assumed to correspond to a variable bound in the top-level or global environment. The manipulation of the top-level environment is omitted since it is neither elucidating nor difficult.

```
(define (compile-IF expr ctenv)
  (let ((pred (compile-expr (cadr expr) ctenv))
        (conseq (compile-expr (caddr expr) ctenv)))
    (if (> (length expr) 3)
        (let ((alt (compile-expr (cadddr expr) ctenv)))
          (LAMBDA (rtenv)
            (if (pred rtenv)
                (conseq rtenv)
                (alt rtenv))))
        (LAMBDA (rtenv)
          (if (pred rtenv)
              (conseq rtenv))))))
```

To compile a conditional (`if`) expression, we compile the subexpressions and collect them into a procedure. This procedure, at run time, will execute the predicate and choose, according to its result, between the consequent and alternative (if present). The metacircular compiler inherits from the underlying system the behavior of an alternative-less `if` expression in which the predicate evaluates to *false*.

```
(define (compile-LAMBDA lam ctenv)
  (let* ((params (cadr lam))
         (body (compile-expr*
                (cddr lam)
                (ct/bind ctenv
                         (if (symbol? params)
                             (list params)
                             params)))))
    (let ((nparams (length params)))
      (case nparams
        ((0)
         (LAMBDA (rtenv)
           (lambda ()
             (body (cons (vector) rtenv)))))
        ((1)
         (LAMBDA (rtenv)
           (lambda (arg)
             (body (cons (vector arg) rtenv)))))
        (else
         (LAMBDA (rtenv)
           (lambda args
             (if (not (= (length args) nparams))
                 (runtime-error "wrong number of arguments" nparams)
                 (body (cons (list->vector args)
                             rtenv)))))))))
```

This code generator for `lambda` expressions handles only `lambda` expressions with a fixed number of bound variables. More complex parameter lists can be handled easily with simple modifications. The procedure generated for a `lambda` expression captures the environment of execution (the closing environment), and returns a procedure that, when invoked, tacks a new environment frame containing the arguments to the captured environment, and then executes the translation of the body of the `lambda` expression in this new runtime environment.[1]

```
(define (compile-combination expr ctenv)
  (let ((oprtr (compile-expr (car expr) ctenv)))
    (case (length expr)
      ((1)
       (LAMBDA (rtenv)
         ((oprtr rtenv))))
      ((2)
       (let ((oprnd1 (compile-expr (cadr expr) ctenv)))
         (LAMBDA (rtenv)
           ((oprtr rtenv)
            (oprnd1 rtenv)))))
      (else
       (let ((oprnds (map (lambda (op)
                            (compile-expr op ctenv))
                          (cdr expr))))
         (LAMBDA (rtenv)
           (apply (oprtr env)
                  (map (lambda (op)
                         (op rtenv))
                       oprnds)))))))))
```

A combination consists of an operator and operands which must be executed, and then the result of the operator is invoked on the results of the operands. The code generator for combinations compiles the operator and operands, and returns a procedure, that, when invoked, invokes all of these compiled subexpressions, and applies the result of the operator to the rest. Since the invocation of the value of the operator is left to the underlying system, and the values of interpreted procedures are

[1]The *optimized* handlers need not check the number of arguments passed because they generate procedures of the correct arity, and the underlying Scheme system presumably takes care of checking the number of arguments.

valid procedures from the underlying system, there is no need to discriminate between primitive, interpreted, and compiled procedures. They are all trivially inter-callable.

Although the code for the compiler is easy to follow, it already includes some *optimizations*. It has special-case code for combinations with fewer than two operands, and lambda expressions with fewer than two formal parameters—it is trivial to extend these to larger numbers, so that the common cases are covered. It would be easy to add improvements such as special-case code for predicates which are combinations of the `not` operator, combinations whose operators are `lambda` expressions (e.g. the expansions of `let` forms), faster access to variables at known lexical addresses, early binding of global procedures, etc.

The correctness of such a compiler should be particularly simple to deduce. Denotational semantics for languages typically involve a similar translation. The meaning function is curried and takes, successively , an expression, an environment, a continuation, and a store.[2] Our compiler is similarly curried, taking an expression, and producing procedures that take concrete environment representations, corresponding to the abstract environments manipulated by the semantics. In essence, our interpreter consists of a denotational semantics interpreter with implicit continuations and stores that it inherits from the implementation language, and whose environments have been made concrete [11, 29, 58].

What may not be obvious is how this code results in a threaded interpreter. Let's revisit the code generated for `if` expressions:

```
(define (compile-IF expr ctenv)
  (let ((pred (compile-expr (cadr expr) ctenv))
        (conseq (compile-expr (caddr expr) ctenv)))
    (if (> (length expr) 3)
        (let ((alt (compile-expr (cadddr expr) ctenv)))
          (LAMBDA (rtenv)
            (if (pred rtenv)
                (conseq rtenv)
```

[2]The continuation and store may not be necessary if the language being described does not have control-transfer operations, or mutation, respectively.

```
                (alt rtenv))))
        (LAMBDA (rtenv)
          (if (pred rtenv)
              (conseq rtenv))))))
```

The result of the compilation of an `if` expression is a closure, a data structure with an entry point and three data fields, containing three other closures (corresponding to `pred`, `conseq`, and `alt`).

This entry point is shared by all such `if` expressions, and is the template code sequence for `if`. It executes the `if` by first invoking (jumping to) the closure for the predicate, and depending on the value computed by that closure, it will invoke either of the other closures corresponding to the consequent or alternative subexpressions.

The machine code corresponding to the template is generated by the native-code compiler when processing the bodies of the `lambda` expressions with an upper-case `LAMBDA` while compiling `compile-if`.

## 4.3 Limitations of the Procedural Representation

As we have seen, we can obtain a threaded interpreter simply by writing a metacircular compiler. Unlike a native threaded interpreter, extending or modifying the language with this interpreter is considerably simpler and less error prone.

Nevertheless, an important reason for having an interpreter in a system that already has a native-code compiler is to increase debugging ability. In addition to the shortened edit-run cycle, one of the reasons that interpreted code can be debugged well is that it is usually not difficult to map the state of a suspended computation into the source forms that produced the state or are pending execution. Yet, if our interpreter is written in the manner described above, this mapping from suspended states to source code is not particularly simple: We need to map the state into the source code of the interpreter by using whatever means our native-code compiler provides, and then invert the metacircular compilation to regenerate the interpreted code's source.

If the compiled procedures used to implement the template code sequences are opaque, we cannot do this conveniently, and we may not be able to debug our interpreted code easily, or at all.

The procedures that we are generating perfectly abstract the behavior of the original expressions under evaluation, but if the procedures are opaque, they abstract it too well. There is no other operation that we can reliably perform on our translated expressions, except to evaluate them by invoking them on a suitable environment. However, we want our expressions not only to be executable, but also to be inspectable, in order to facilitate debugging. Opaque procedures eliminate our ability to do this.

As in previous scenarios, there are distasteful and fragile tricks that we might use to solve this problem. For example, we can change the interface to our procedures to maintain a dual representation, so that when given a recognizable argument, they will return the source expression instead of executing it.

We can therefore, change our code uniformly, as suggested by the following example:

```
(define (compile-IF expr ctenv)
  (let ((pred (compile-expr (cadr expr) ctenv))
        (conseq (compile-expr (caddr expr) ctenv)))
    (if (> (length expr) 3)
        (let ((alt (compile-expr (cadddr expr) ctenv)))
          (LAMBDA (rtenv)
            (if (eq? rtenv 'DISASSEMBLE)
                `(if ,(pred rtenv)
                     ,(conseq rtenv)
                     ,(alt rtenv))
                (if (pred rtenv)
                    (conseq rtenv)
                    (alt rtenv)))))
        (LAMBDA (rtenv)
          (if (eq? rtenv 'DISASSEMBLE)
              `(if ,(pred rtenv)
                   ,(conseq rtenv))
              (if (pred rtenv)
                  (conseq rtenv)))))))
```

Unfortunately this obscures our code, and slows it down because of the tests at every step.

Alternatively, we can keep a table of associations between these procedures and the original expressions, which we can query as necessary. This is adequate in certain circumstances, but new entries will have to be added to this table every time that a procedure is created, which occurs when `lambda` expressions are executed. The cost of evaluating `lambda` expressions grows noticeably in our attempt to keep the table up to date, and, of course, the compiler must be peppered with extraneous code used to maintain the table.

Some of these tricks are workable, but even if they impose no efficiency loss, they are undesirable. A very important property of this technique for constructing interpreters is that the code is so simple and directly matches the semantics. Additional code, to enable us to invert our translation, would only clutter our interpreter/compiler, and should not be necessary.

## 4.4 Overcoming Opacity

As before, we can overcome the invertibility problem by using translucent procedures. If the procedures used to represent and implement our interpreted code can be decomposed, we can invert the translation and reconstruct the source.

To carry this out, in the code for the compiler, we can replace `lambda` with `tlambda` in the `lambda` expressions that yield the procedures that our interpreter manipulates.[3][4][5]

For example, `compile-if` becomes

```
(define (compile-IF expr ctenv)
  (let ((pred (compile-expr (cadr expr) ctenv))
```

[3]Precisely the upper-case `lambda`s.

[4]TScheme is not applicative-order, but this could be changed.

[5]TScheme does not have variable-arity procedures, so the last clause of `compile-lambda` would have to be rewritten in terms of `tproc/make`.

```
            (conseq (compile-expr (caddr expr) ctenv)))
        (if (> (length expr) 3)
            (let ((alt (compile-expr (cadddr expr) ctenv)))
              (tlambda (rtenv)
                (if (pred rtenv)
                    (conseq rtenv)
                    (alt rtenv))))
            (tlambda (rtenv)
              (if (pred rtenv)
                  (conseq rtenv))))))
```

Once this is done, besides using `tproc/decompose` directly, we can use our procedural pattern matcher to destructure the output of the compiler. For example, we can easily determine whether a procedure corresponds to an `if` expression by using the following predicate:

```
(define (compiled-IF-expression? expr)
  (or (match? (tlambda (rtenv)
                (if (#?PRED rtenv)
                    (#?CONSEQ rtenv)
                    (#?ALT rtenv)))
              expr)
      (match? (tlambda (rtenv)
                (if (#?PRED rtenv)
                    (#?CONSEQ rtenv)))
              expr)))
```

Similarly, we can reconstruct the source by using

```
(define (invert-compiled-IF expr ucenv)
  (let ((decode
         (lambda (result unk)
           (invert (match/lookup result unk) ucenv))))

    (cond ((match? (tlambda (rtenv)
                     (if (#?PRED rtenv)
                         (#?CONSEQ rtenv)
                         (#?ALT rtenv)))
                   expr)
           => (lambda (result)
                `(IF ,(decode result #?PRED)
                     ,(decode result #?CONSEQ)
                     ,(decode result #?ALT))))
          ((match? (tlambda (rtenv)
```

```
                    (if (#?PRED rtenv)
                        (#?CONSEQ rtenv)))
                  expr)
          => (lambda (result)
               `(IF ,(decode result #?PRED)
                    ,(decode result #?CONSEQ))))
         (else
          (error "invert-compiled-IF: Not an IF expression"
                 expr)))))
```

Note the correspondence between the code generated by `compile-if`, and the patterns in `invert-compiled-if`. If a new template is introduced for IF expressions in `compile-if`, `invert-compiled-if` may have to be extended in a similar manner to allow the reconstruction of the source code for the newly generated templates.

As in the case of byte-coded interpreters, the ability to reconstruct expressions into the original source depends on how the expressions are compiled. For example, we can compile `or` forms using the native-code compiler's `or` keyword, or we can macro-expand them into expressions using `let`, `if`, and `lambda`, and compile the result. The second choice makes the translations of `or` expressions and the corresponding expansions indistinguishable, so our inverter will be unable to invert them properly.

```
(define (compile-OR expr ctenv)
  (if (null? (cddr expr))
      (compile-expr (cadr expr) ctenv)
      (let ((pred (compile-expr (cadr expr) ctenv))
            (rest (compile-OR `(OR ,@(cddr expr)) ctenv)))
        (tlambda (rtenv)
          (or (pred rtenv)
              (rest rtenv))))))

(define (compile-OR expr ctenv)
  (define (macro-expand-OR expr)
    (if (null? (cdr expr))
        (car expr)
        `(let ((pred ,(car expr))
               (rest (lambda ()
                       ,(macro-expand-OR (cdr expr)))))
           (if pred
               pred
               (rest)))))
```

```
(compile-expr (macro-expand-OR (cdr expr))
              ctenv))
```

An additional problem with respect to invertibility is the following: The generated code does not need the names of lambda-bound variables in the code being translated. The variable names have been translated into code sequences that directly implement the lexical address lookup for the variables, and names are only necessary (if at all) for variables not visible in the compile time environment.[6] Hence our inverter will not be able to recover the names and may have to generate them. A production version of the metacircular compiler should try to keep these names in order to better invert code.

Invertibility is not only useful for debugging, but also to simplify implementations. The interpreter/compiler that we show is somewhat incestuous, since it interprets and is written in Scheme; the whole approach is predicated on the prior existence of a native-code compiler to bootstrap and accelerate the code.

Consider instead the possibility of implementing a *different* language using the same techniques. We may want to provide both an interpreter and compiler for the new language. In order to use the pre-existing Scheme native code compiler, we can consider source-to-source translation as the basis of our implementation. We can implement a source-level translator from our new language to Scheme, and then use either the Scheme interpreter, or the Scheme compiler to execute the code. Although this produces a working implementation, we are better off using a compiler that directly produces procedures such as those presented above. The resulting interpreted code is faster, since it directly manipulates the runtime data structures of our new language, instead of Scheme environments emulating the manipulation of these data structures. The ability to extract Scheme code from our resulting procedures (by using `tproc/decompose`) allows us to use the Scheme native code compiler to gain

[6]Translating variable references to lexical addresses is analogous to using variable-free de Bruijn notation for the $\lambda$-calculus [7, 10].

performance, while concentrating all the knowledge about the semantics of our new language in the translator to procedures.

## 4.5 Summary

Using closures (procedures) to implement interpreters leads to clearly written and efficient interpreters, where the meaning of an expression or statement has been captured precisely by the procedure that implements it. However, an important property of interpreters is that the code executed can be easily mapped to the source code. Ordinary procedures make this task difficult because of their opacity. Translucent procedures allow the inspection necessary to perform this mapping, while keeping the elegant code for the interpreter unchanged.

# Chapter 5

# Constructive Non-elementary Functions

The final scenario that we will explore is the construction of non-elementary mathematical functions, which can be represented naturally and easily with procedures.

Non-elementary (e.g. transcendental) functions can be constructed from definitional properties by using procedural composition. In this chapter we will see how this can be done and why opaque procedures preclude the efficient execution functions constructed in this way. Translucent procedures allow us to invert the resulting combination into expressions that can be optimized and compiled.

## 5.1 Constructing functions from their Defining Properties

Gerald Roylance shows [52] how many standard mathematical functions can be constructed from their defining properties, instead of specified as a sequence of obscure arithmetic computations. Constructing mathematical subroutines—rather than just providing them—is very powerful: different applications may need different basis functions for expansion, higher precision, or use a different numeric representation.

Furthermore, constructed routines are not subject to coefficient transcription errors, and are portable to systems with different floating-point characteristics.

The essence of Roylance's method is to use the mathematical properties of a function to produce an implicit exact representation of it. The representation is then truncated, and the values involved are perturbed (using standard numerical analysis techniques) yielding an efficient implementation for the desired numerical representation and accuracy. All of these steps are described by programs, rather than carried out by hand.

For example, analytic functions can be described in terms of their Taylor series, which can be exactly represented, although implicitly, by a function that generates successive coefficients. The series can be truncated at a point where the error is acceptable for the desired level of accuracy, and Chebyshev economization [13] can be employed on the resulting finite polynomial to obtain a new one that can be used to compute more efficiently. This operation has traditionally been carried out by hand, producing a table of coefficients that are then explicitly coded into the subroutines that implement floating-point transcendental functions. Because of the possibility of error in this computation, these coefficients are often collected in mathematical handbooks, but transcription errors are not unknown.

Once given a list of coefficients, in order of ascending degree, it is simple to construct a procedure that will compute the desired approximation by using Horner's rule [39, 1] for evaluating polynomials:

```
(define (coeffs->fcn coeffs)
  (let ((descending (reverse coeffs)))
    (lambda (x)
      (coeffs-eval descending x))))

(define (coeffs-eval coeffs x)
  (let loop ((val 0) (coeffs coeffs))
    (if (null? coeffs)
        val
        (loop (+ (car coeffs)
                 (* x val))
```

```
              (cdr coeffs)))))
```

A sine routine that operates on the interval $[0, \frac{\pi}{2}]$ might be constructed as follows:

```
(define sine-half
  (coeffs->fcn
   (let ((eps 1.e-16))
     (chebyshev-economization-scaled
      (/ pi 2)
      (truncated-series-eps sine-term
                            sine-mono
                            eps
                            (/ pi 2))
      eps
      (* eps 10)))))
```

## 5.2 Construction by Abstraction and Composition

There are several inefficiencies in the code presented above that makes this method compare unfavorably with traditional techniques:

- There is overhead in traversing the list holding the coefficients. A large part of the run time of `coeffs-eval` is due to traversing the coefficient list, even though any given function generated by `coeffs->fcn` has a constant list.

- The floating-point register and pipeline are likely to be under-utilized because of the *rolled* loop in `coeffs-eval`. For any given function generated by `coeffs->fcn`, there is an optimal unrolling of the loop in `coeffs-eval` that exposes all of the temporaries and operations to a register allocator and instruction scheduler, but `coeffs-eval` is only unrolled a particular number of times, if at all.

- As seen below, coefficients may end up being zero or one, yet `coeffs-eval` will blindly add zero and multiply by one.

- There is no redundant subexpression elimination in the resulting code. When the computation is fully unrolled, and the coefficients examined, there might be redundant common subexpressions that would save floating-point operations and registers if they were properly removed.

For example, `sine-half` constructed above might correspond to the same arithmetic operations as

```
(lambda (x)
  (+ 0
     (* x
        (+ .9999999999999992
           (* x
              (+ 0
                 (* x
                    (+ -.16666666666664778
                       (* x
                          (+ 0
                             (* x
                                (+ 8.333333333226133e-3
                                   (* x ...)))))))))))))
```

but the additions of zero will be carried out, and the common subexpression `(* x x)`—due to the function being odd—will be evaluated many times, rather than just once. Overall, half of the floating-point operations and all the list destructuring are avoided by a traditional definition.

We can eliminate some of these sources of inefficiency by using a more procedural approach, reminiscent of the metacircular compiler. We can rewrite `coeffs->fcn` in the following way:

```
(define (coeffs->fcn coeffs)
  (if (null? coeffs)
      (const->fcn 0)
      (+fcn (const->fcn (car coeffs))
            (*fcn identity
                  (coeffs->fcn (cdr coeffs))))))

(define (const->fcn c)
  (LAMBDA (x)
```

```
    c))

(define identity
  (LAMBDA (x)
    x))

(define (+fcn f g)
  (LAMBDA (x)
    (+ (f x)
       (g x))))

(define (*fcn f g)
  (LAMBDA (x)
    (* (f x)
       (g x))))
```

This eliminates the runtime conditionals that determine the structure of the list, and we can easily implement some optimizations to avoid multiplying by one, or adding zero. For example, we can use the following code instead:

```
(define (coeffs->fcn coeffs)
  (if (null? coeffs)
      (const->fcn 0)
      (term (car coeffs)
            (coeffs->fcn (cdr coeffs)))))

(define (term val rest)
  (if (zero? val)
      (LAMBDA (x)
        (* x (rest x)))
      (LAMBDA (x)
        (+ val
           (* x (rest x))))))
```

## 5.3 Closures are not Good Enough

Unfortunately, neither of the prior versions that compose procedures provides a real improvement in efficiency with respect to the original version that traversed the list of coefficients every time that the resulting function was invoked. We have replaced traversing the list of coefficients in a tight loop with traversing a computed call graph at run time. Furthermore, even though we can eliminate some of the more glaring inefficiencies, we cannot use the power and convenience of a full-fledged algebraic simplifier that is likely to do a much better job, since it need not restrict itself to local optimization.

Alternatively, we can proceed by generating not procedures, but `lambda` expressions to be handed to a compiler or `eval`. We could even invoke an algebraic simplifier on the resulting expression before handing it to our compiler or evaluator. However, shifting to the domain of expressions complicates the problem because of additional extraneous details including the names of the bound variables and operations, and the generation of syntactically correct code. In addition, unless we use a native-code compiler, the result may well be slower than the version provided above. Finally, invoking a native-code compiler for functions generated and discarded quickly is not attractive—The cost of the compilation may not be amortized over the number of calls to the result.

A partial evaluator [19] can alleviate some of our problems—we can specialize our polynomial evaluator for each of the common non-elementary functions (sine, exponential, bessel, etc.). Yet, although partial evaluation reduces the overhead of traversing lists or closures, it does not, by itself, give us a way to insert an algebraic simplifier or a redundant subexpression eliminator into the picture. One further disadvantage of partial evaluators is that they are not well suited for dynamic generation of functions. Not only must the code for the partial evaluator be present in our running system, but they also require an evaluator or compiler to process the result, which must be resident as well.

Ultimately, the procedural representation has advantages. The generation is quick even if the result is not very fast. If the result is going to be used only a few times, or infrequently, the code above is probably adequate. However, it is not acceptable for common routines, such as `sin`, `exp`, or `sqrt`.

One might be tempted to have two versions of the code. One for dynamic generation of functions that will be used infrequently, the other for those that will be used more frequently, perhaps by using macros or other source transformation. This duplication is obviously undesirable.

Although we can improve the procedural representation by using tricks such as the one used earlier, this becomes progressively more difficult. Our optimizer should not be interleaved with our constructor because it makes both tasks more error prone. Our constructor should always generate correct—even if slow—procedures, and we should have the ability to optimize them when and if it is convenient. However, this is difficult if our language only provides opaque procedures, unless we resort to tricks similar to those considered when discussing the equation solver.

## 5.4 Having Our Cake and Eating It Too

As in the case of the equation solver, translucent procedures allow us to construct our functions by using composition, but also to take the result apart, not only to discriminate between alternatives, but to optimize the result. If we write our combinators using `tlambda` instead of `lambda`, then we can inspect the result. For example, `*fcn` would be written as follows.

```
(define (*fcn f g)
  (tlambda (x)
    (* (f x)
       (g x))))
```

Using the procedural matcher, we can write an optimizer, but this is not the best solution. Expressions are easier to manipulate, and can then be given to a native code compiler. Translucent procedures allow us to convert between procedures and expressions by using procedures similar to `fcn->expression`:

```
(define (fcn->expression fcn var)
  (define (inner fcn)
    (cond ((match? (tlambda (x) (+ #?CONST (* x (#?NEXT x))))
                   fcn)
           => (lambda (result)
```

```
                  `(+ ,(match/lookup result #?CONST)
                      (* ,var
                         ,(inner (match/lookup result #?NEXT)))))
            ((match? (tlambda (x) (* x (#?NEXT x)))
                     fcn)
             => (lambda (result)
                  `(* ,var ,(inner (match/lookup result #?NEXT)))))
            ((match? (tlambda (x) #?CONST)
                     fcn)
             => (lambda (result)
                  (match/lookup result #?CONST)))
            (else
             (error "fcn->expression: Unknown pattern" fcn))))

  (inner fcn))
```

Now

```
(fcn->expression (coeffs->fcn '(1 0 1 0 1)) 'y)
```

returns the list

```
(+ 1 (* y (+ 0 (* y (+ 1 (* y (+ 0 (* y (+ 1 0)))))))))
```

which can be processed by an algebraic simplifier, or a common subexpression eliminator, and then given to a native code compiler:

```
(define (optimize-function p)
  (compile-lambda
   `(lambda (y)
      ,(simplify (fcn->expression p 'y)))))
```

## 5.5 Summary

Mathematical functions can be constructed easily from definitional properties by procedure composition and abstraction. However, the result is not as efficient as a hand-coded version. In order to obtain the same efficiency from our generated versions as from those obtained by traditional methods, we need to be able to convert the results into alternate representations, better suited for simplification and compilation.

Different operations are often performed best by using different data representations. Large programs frequently convert their data from one representation to another, in order to better operate on it. Such programs cannot use opaque procedures to represent data, because there is no way to then transform them into alternate representations, better suited to other tasks.

Translucent procedures overcome this problem. Procedures are a good representation for those parts of a computation in which data captures a behavior, and where it is the execution and composition of this behavior that matters. Unlike opaque procedures, translucent procedures additionally allow the translation to different representations when different properties of the data become important.

# Chapter 6

# Semantic Concerns

In preceding chapters, we have used a language in which procedures can not only be invoked, but also operated upon by unusual new primitives. We must consider implications translucent procedures have for the semantics of programs.

## 6.1 A Trivial Semantics for Translucent Procedures

It is an easy task to write a denotational semantics [58, 29] for a Lisp-like language with operations such as `tproc/make` and `tproc/decompose`.

Procedures in Lisp-like languages are represented by functions in their semantics.[1] The meaning of a procedure call, or invocation of a procedure to arguments, is the value of the function denoted by the operator at the denotations of the arguments, with an appropriate continuation and store.

In the absence of variable assignment, `tproc/make` implies no additional complexity. Its two arguments are the explicit representation of a `lambda` expression, and the explicit representation of an environment, say as an association list pairing

[1] In the formal semantics for Scheme, procedures are represented as a pair of a location and a function in order to support `eq?` and `eqv?` on procedures.

symbols with objects. The meaning of a call to `tproc/make`, if the arguments are of appropriate types, is simply the same as the meaning of the evaluation of the internal representation of its first argument in the environment represented by the second argument. Besides introducing conversion functions to map between external representation of expressions and environments and the internal ones no other change is required in the semantics.

Adding `tproc/decompose` is somewhat more complicated, but hardly difficult. The simplest way to add this operation to our formal semantics is to change the representation of compound procedures (not primitives). Compound Procedures can now be represented as triples containing an expression, an environment, and the function we would have used otherwise. The expression is the `lambda` expression whose evaluation resulted in the procedure. This evaluation may have been implicit, or explicitly requested through the use of `tproc/make`. The environment is the environment function where the `lambda` expression was evaluated. Procedure invocation ignores the expression and environment components of the triple, and uses the function in the same way as in the unextended semantics.

We only need to describe `tproc/decompose` itself. `tproc/decompose` returns an expression and an environment, as a pair, or as two separate values. In the simplest implementation, it merely extracts the expression and environment components from the representation of its argument, maps them to their external representation, and returns them.

The expression component is transformed into *ordinary* data structures by a simple recursive walk.

Mapping the environment function into a data structure is a little more complicated. The environment function maps each identifier, typically from a countable infinite set, to a value, and we cannot directly represent such an infinite data structure. However, most of the identifiers map to $Undefined$, an undefined value, and the only identifiers that matter are those that are referenced freely by the expression.

It is straightforward to compute the free variables of a finite expression, map them to values using the environment function, and collect only those in the data structure.

## 6.2 Transparency Reveals Too Much

A semantics modified as outlined above, and fully specified in appendix B fulfills our contract, but is unsatisfactory because it is too concrete.

The purpose of a formal semantics is not only to define a language without reference to a particular implementation, but also to provide a formal framework for proofs about the effects and *equivalence* of programs. This issue is not only of theoretical significance; A large part of the work of a compiler, particularly for higher-order languages, is to rephrase programs, or pieces of programs, into equivalent, more efficient versions.

Unfortunately, the very simple modification that we have introduced has profound consequences. Programs that formerly had the same meaning—perhaps even provably so—may cease to be equivalent because the actual expressions captured in the procedures differ. This affects not only programs that use the new features, but all programs—the change in procedure representation is pervasive! Of course, we can have two different proof systems (and semantics), one for programs that use the newly introduced features and one for those that do not, but this is undesirable. Program fragments, examined in isolation, would almost invariably have to be examined through the lens of the richer (and tighter) semantics.

Happily, the situation is not as bad as it might appear at first. Even if the expressions (and environments) captured by procedures make two procedures distinct, if we can prove that their meaning under invocation is identical, that is, that their function components are equal—we can still substitute one for the other when invoked, or decide that the values returned from such invocations are the same. This is a common occurrence, since first-order code can be analyzed with the unmodified semantics.

However, we can ameliorate the problem in the semantics. The problem is that the *original* expression and environment that resulted in a procedure are too specific, although they clearly satisfy the specification. The ability to obtain the original expression not only complicates analysis, and prevents our compiler from optimizing some code, but it is not quite what we want or need. If the main purpose of this ability is to be able to compare the representations of procedures for equivalence of meaning, or to deduce what various components must be in order to make the meaning the same, then this ability is getting in our way, since the expressions that our semantics makes available have now become part of the meaning of our programs.

## 6.3 Translucency Reveals What We Can Use

We want to obscure some of the detail of the representation (and history) of our procedures, without giving up fully, making them opaque, or preventing the creation of tools such as the pattern matcher.[2]

Our real requirement is that `tproc/decompose` must return an expression and environment that, when given to `tproc/make`, will construct a procedure that will behave as the original. In other words, if we refer to the composition of `tproc/make` and `tproc/decompose` by the name `tproc/id` we want the following identity to hold

$$\mathcal{E}[\![\mathtt{E}]\!]\rho = \mathcal{E}[\![(\mathtt{tproc/id\ E})]\!]\rho$$

whenever $IsProcedure?(\mathcal{E}[\![\mathtt{E}]\!]\rho)$ is true.

Our initial implementation satisfies this trivially, by, making `tproc/make` and `tproc/decompose` simple inverse representation changes. However, the trivial solution is neither interesting, nor, as we have discussed, desirable.

How do we solve this problem? If we do not want our new features to interfere with our ability to decide when two programs are the same, we can define our new

[2] It is this partial opacity that makes these procedures translucent, rather than transparent.

operations taking our equivalence predicate into account. We can partly obscure the expression and environment that form part of our procedures so that our semantics no longer distinguishes among the programs that we desire to be equivalent.

Rather than simply collecting the expression and environment into a procedure when evaluating a `lambda` expression, or invoking `tproc/make` on their external representations, we apply a canonicalization function to the representation of the expression and environment. This canonicalization function must be invariant under our equality predicate, and the resulting procedure contains the canonicalized representations, which `tproc/decompose` can extract. Since the meaning of these alternatives is the same once more, any theorems that we might prove using our equality predicate in the unextended semantics, are still valid. Similarly we may perform any substitutions that we can derive from our equality predicate, since the canonicalization function prevents us from distinguishing among the alternatives in the extended semantics.

This agenda presumes the existence of a (computable) canonicalization function for program representations, a question that we must address. The remainder of this chapter is an informal proof that such functions always exist.

## 6.4 Preliminary Concepts

Fix a finite set of distinguishable simple primitive procedures such as `cons`, `+`, etc. By simple we mean that they do not manipulate procedures or environments, nor do they create infinite[3] or cyclic values.

In the following discussion, we need the following concepts:

- The *support* of an environment is the set of identifiers that are not mapped to $Undefined$. Consequently, an environment has *finite support* if it maps all but a finite number of identifiers to $Undefined$.

[3] "Infinite" is used informally here, not as defined below.

- An environment is a *base environment* if it has finite support and maps identifiers only to $Undefined$ or to opaque, distinguishable primitive procedures. A base environment models the initial environment where whole programs are evaluated and contains a finite number of bindings for primitives procedures such as `+`, `cons`, and `tproc/make`. The empty environment, mapping all identifiers to $Undefined$, is a base environment.

- An expressible value is a *finite value* if it is not $\perp$ and is the meaning of some finite expression in some base environment. A finite value is the result of some finite computation.

- An environment is a *finite environment* if it has finite support and only binds identifiers to $Undefined$ or finite values.

**Lemma 6.1** *The value of a finite expression in a finite environment is $\perp$ or a finite value.*

Proof: A finite environment $\rho$ binds a finite number of identifiers to a finite number of finite values. Each such value, by definition, can be represented as a pair of a finite expression and a base environment.

Therefore there exist a positive integer $n$, $n$ identifiers ($\mathtt{Id}_1 - \mathtt{Id}_n$), $n$ finite expressions ($\mathtt{E}_1 - \mathtt{E}_n$), and $n$ base environments ($\rho_1 - \rho_n$), with $\mathcal{E}[\![\mathtt{E}_i]\!]\rho_i \neq \perp$, such that

$$\begin{array}{lllll} \rho & = & \lambda \;\; I. & (I = \mathtt{Id}_1) & \rightarrow \;\; \mathcal{E}[\![\mathtt{E}_1]\!]\rho_1, \\ & & & (I = \mathtt{Id}_2) & \rightarrow \;\; \mathcal{E}[\![\mathtt{E}_2]\!]\rho_2, \\ & & & \ldots & \\ & & & (I = \mathtt{Id}_n) & \rightarrow \;\; \mathcal{E}[\![\mathtt{E}_n]\!]\rho_n, \\ & & & Undefined & \end{array}$$

Without loss of generality the $\rho_i$ bind distinct identifiers. Let $\rho'$ be the *union* of all these environments, i.e. the function whose support is the union of the supports of the $\rho_i$, and that maps each identifier in its support to the unique non-$Undefined$

value that some $\rho_i$ maps it to. Clearly $\rho'$ is a base environment, since its support is the finite union of finite sets.

Now, for any finite expression E, let $v$ be defined as follows.

$$v = \mathcal{E}[\![\mathtt{E}]\!]\rho = \mathcal{E}[\![((\mathtt{lambda}\,(\mathtt{Id}_1\ \mathtt{Id}_2 \ldots \mathtt{Id}_\mathtt{n})\ \ \mathtt{E})\quad \mathtt{E}_1\ \mathtt{E}_2 \ldots \mathtt{E}_\mathtt{n})]\!]\rho'$$

But $v$ is either $\perp$ , or, clearly, the value of a finite expression in a base environment, i.e. a finite value.

**Lemma 6.2** *Lemma 6.2: Finite values and finite environments are finitely representable.*

Proof: Consider the finite alphabet from which expressions are constructed. Extend it by adding a new symbol for each primitive in our finite fixed set (e.g. **#+** for **+**, **#CAR** for car, etc.), as well as a few punctuation symbols (e.g. **<,>,=**).

- Base environments are finitely representable as the punctuated concatenation of the finite number of identifiers in their support followed by the unique symbols corresponding to the primitives.

  E.g. $\rho \mapsto$ <FOO=#CAR,BAR=#+>

- Finite values are finitely representable as the punctuated concatenation of the finite expression and the base environment.

  E.g. $3 \mapsto$ <(lambda (x) (+ x 3)),<+=#+>>.

- Finite environments are finitely representable as the punctuated concatenation of the finite number of identifiers in their support followed by finite representations of the values.

  E.g. $\rho \mapsto$ <FOO=<(lambda (x) (+ x 3)),<+=#+>>,BAR=<(lambda (y) (- y 7)),<-=#->>>.

This representation is clearly not unique; Not only can the bindings in an environment be reordered, but the pair of expression and environment for finite values is not unique either.

Lemma 6.2 only shows that finite representations exist. A semantics can compute such finite representations by extending all expressible values to carry their representations, using the rewrite that appears in the proof of lemma 6.1 in its operation. However, this particular representation is expensive to compute, and not terribly useful. In addition, on languages with locations, the locations provide simpler ways to represent finite values.

## 6.5 Computable Canonicalization Functions

We can now tackle equality of program representations. A program representation consists of the representation of an expression and the representation of an environment. An equality predicate on program representations is simply any reflexive, symmetric, and transitive relation on such pairs. However, it is more convenient to view such relations as their characteristic functions; that is, as boolean functions (true if related, false if not) of four arguments. We can say that an equality predicate is compatible with a semantics if the predicate considers two programs equivalent only when their meanings are identical.

In other words, an equality predicate $Eqv$ is compatible with a semantic function $\mathcal{E}$ if the following holds for representations $\mathtt{E}_1$, $\mathtt{E}_2$, $\mathtt{Rho}_1$, and $\mathtt{Rho}_2$.

$$Eqv(\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2) \Rightarrow \mathcal{E}[\![e_1]\!]\rho_1 = \mathcal{E}[\![e_2]\!]\rho_2$$

where $e_1$, $\rho_1$, $e_2$, and $\rho_2$ are the expressions and environments being represented by $\mathtt{E}_1$, $\mathtt{E}_2$, $\mathtt{Rho}_1$, and $\mathtt{Rho}_2$ respectively.

Of course, this is equivalent to

$$\mathcal{E}[\![e_1]\!]\rho_1 \neq \mathcal{E}[\![e_2]\!]\rho_2 \Rightarrow \neg Eqv(\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2)$$

To claim that our *extended* semantics, with semantic function $\mathcal{E}_e$, is not a serious perturbabion of our *unextended* semantics, with semantic function $\mathcal{E}_u$, we would like the following to hold:

$$\mathcal{E}_u[\![e_1]\!]\rho_1 = \mathcal{E}_u[\![e_2]\!]\rho_2 \Rightarrow \mathcal{E}_e[\![e_1]\!]\rho_1' = \mathcal{E}_e[\![e_2]\!]\rho_2' \tag{6.1}$$

Where $\rho_1'$ and $\rho_2'$ are the environments corresponding to $\rho_1$ and $\rho_2$ but defined on the domains of our extended semantic function $\mathcal{E}_e$.

Thus we want to use an equality predicate that is compatible with our *unextended* semantic function $\mathcal{E}_u$. Ideally, we would like to use the following equality predicate:

$$Eqv * (\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2) \stackrel{\text{def}}{=} (\mathcal{E}_u[\![e_1]\!]\rho_1 = \mathcal{E}_u[\![e_2]\!]\rho_2) \tag{6.2}$$

But this predicate is, of course, undecidable, and no compiler or mechanical deduction system will be able to implement it.

Thus we cannot really satisfy condition 6.1, but we can come close. Since our ideal goal, $Eqv*$, defined in equation 6.2, is undecidable, mechanical program analysis and manipulation tools cannot use it and must approximate it.

It is reasonable, therefore, to restrict ourselves to the set of *decidable*, equality predicates compatible with our unextended semantics, and also to finite expressions and finite environments, which, by lemma 6.2, are finitely representable. The set of decidable equality predicates compatible with our semantics on finite expressions and environments is clearly not empty. The trivial equality predicate $Eqv_0$,

$$Eqv_0(\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2) \stackrel{\text{def}}{=} ((\mathtt{E}_1 \doteq \mathtt{E}_2) \bigwedge (\mathtt{Rho}_1 \doteq \mathtt{Rho}_2))$$

where $\doteq$ means identity of the representations (e.g. string equality), is both decidable, and clearly compatible with our unextended semantics.

Other examples of acceptable equality predicates are equality modulo $\alpha$-conversion (arbitrary, consistent renaming of bound variables), and equality after finite unfoldment.[4]

Of course, sharper decidable equality predicates for our unextended semantics will lead to better approximations of condition 6.1.

[4]To make the predicate reflexive, we still need to check for identity.

Given any equality predicate, we can consider the set of its canonicalization functions. A function from programs (pairs of representations of expressions and environments) to programs is a canonicalization function if it maps every member of an equivalence class in our predicate to a unique program in the class. An equality predicate partitions the set of programs into equivalence classes, and a canonicalization function chooses a representative from the class. In other words, $Can$ is a canonicalization function for $Eqv$ if and only if it is defined on every program, and the following two implications hold:

$$\begin{aligned} Can(\mathtt{E}_1, \mathtt{Rho}_1) = (\mathtt{E}_2, \mathtt{Rho}_2) \quad &\Rightarrow \quad Eqv(\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2) \\ Eqv(\mathtt{E}_1, \mathtt{Rho}_1, \mathtt{E}_2, \mathtt{Rho}_2) \quad &\Rightarrow \quad Can(\mathtt{E}_1, \mathtt{Rho}_1) \doteq Can(\mathtt{E}_2, \mathtt{Rho}_2) \end{aligned}$$

for all $\mathtt{E}_1$, $\mathtt{E}_2$ representations of expressions, and $\mathtt{Rho}_1$, $\mathtt{Rho}_2$ representations of environments, and where $\doteq$ is identity of the representations (e.g. string equality).

**Lemma 6.3** *For every* decidable *equality predicate compatible with our semantics, there exists at least one* computable *canonicalization function.*

The proof is straightforward: Given a program, finitely represented as a string in some fixed alphabet, we can generate all strings of the same or smaller length, remove those that are syntactically invalid, sort them from shorter to longer, and within the same length lexicographically, and compare each one to our original program using our equality predicate. The first in the sequence found to be equivalent is our answer. Since the predicate is reflexive, and the program itself is in the sequence, there is always at least one program equivalent to the input in the set considered. The algorithm terminates because our predicate is decidable and we only have a finite number of strings to test. The algorithm is a canonicalization program because given any two equivalent programs, the sequence of strings examined for one of them is an initial prefix (perhaps identical) of the sequence of strings examined for the other, and the transitivity of the equality predicate guarantees that both will find the same first equivalent string.

## 6.6 Summary

We show that decidable equality predicates compatible with our unextended semantics always exist, and that each decidable equality predicate has at least one computable canonicalization function.

We can use such a predicate and function in our extended semantics to partly obscure the representation and history of procedures. In the extended semantics, the expression and environment components of a procedure are canonicalized, and `tproc/decompose` produces the external representation of these obscured values.

Compilers and other mechanical program analyzers can now equate procedures whose denotations are the same, and this predicate is, by restriction, decidable and compatible with the original semantics. As desired, compilers and other tools can optimize (rephrase) programs as long as the result is compatible with the equality predicate.

While the language definition can fix (for all time) such an equality predicate and canonicalization function, this is unnecessary, and prevents improvements in program proof technology from being included in compilers and other such tools. Alternatively, the predicate and canonical function can remain unspecified, but restricted as outlined above. The compiler writer then has a free hand, but it is desirable, under these circumstances, that tools such as the pattern matcher use an equality predicate that subsumes what the compiler uses.

Clearly, in a real system, expressions and environments need not be canonicalized when assembled into procedure objects, but can be canonicalized when extracted by `tproc/decompose`. Furthermore, the canonicalization function can be implemented within the language, since it is computable, and `tproc/decompose` can merely use it on the result of a *sub-primitive* that returns the original expression and environment just like our original version of `tproc/decompose`.

In fact, the canonicalization need never be performed at all. It is more useful to provide the equality predicate as a primitive. The canonicalization function is really just a trick to make our semantics determinate, restoring the freedom to our mechanical analysis tools that the reification of procedures apparently withdrew.

# Chapter 7

# Efficiency concerns

When considering a feature for inclusion in a programming language one must examine it under different perspectives. The most common criteria are:

- Expressive power
- Semantic cleanliness
- Efficiency

When translucent procedures are added to a language, certain programs can be constructed in a manner that is clear, elegant, and effective. If these same programs are written without the benefit of the aforementioned extensions to the base language, the resulting construction process is more difficult and obfuscated. In the immediately preceding chapter we argue that the addition of translucent procedures does not unduly complicate the semantics of a language; in particular, our ability to write program analysis programs, such as compilers, is not diminished, because we can restrict ourselves to decidable algorithms, which are the best our mechanical tools will ever be able to use. Thus, the only concern remaining is the question of efficiency.

The efficiency question can be divided into three different components:

1. The cost to programs that use the new feature.

2. The effect of the *existence* of the new feature on the efficiency of programs that do not use it.

3. The effect of the *existence* of the new feature on the complexity of implementations of the language.

All of these aspects are important. If a feature is very expensive on its own, programmers will avoid it, and adding it to the language will not be very profitable. If the mere existence of a feature significantly penalizes most other programs, its addition should be questioned. Although programming languages should be viewed fundamentally as formal languages for the communication of *how to* information (rather than *what is*, as mathematics), machine execution should not be dismissed. Machines can be used not only to accomplish a task, but to test programs on a well-chosen finite number of inputs to confirm correctness, or, more exactly, to reject incorrectness. A severe efficiency penalty makes either of these possibilities less practical, and programmers often opt for a more efficient, even if less expressive, language. Finally, if the addition of a feature complicates the task of implementing the language to the point where it becomes effectively unimplementable, or very difficult to implement, implementations will be rare, and due to the complexity, probably both error-prone and inefficient, thus preventing the actual use of the language.

Let us address question three first. The cost of translucent procedures in terms of implementation complexity depends on implementation technique. For simple interpreters, the cost is very low. `tproc/decompose` (or the sub-primitive on which it is based) merely needs to return a representation of the `lambda` expression that resulted in a procedure, and a representation of the environment. Interpreters usually maintain environments in very regular ways, and it should be easy to perform the translation. Furthermore, it does not matter whether the interpreter directly executes the original expression, or it executes the expression translated to a different language (e.g. byte code). Byte codes can be made invertible, and the function mapping source programs to byte code programs need not be a bijection. After all, the expression and environment returned by `tproc/decompose` need not be the original, merely ones that provably have the same behavior.

As the implementation introduces more and more optimizations—or alternatively, translates the code into native machine language—the executed instructions become difficult to invert into source expressions; we reach a point when it is advantageous to keep the original expression (perhaps at some stage of its translation), and associate it with the object code, so that it can be recovered by `tproc/decompose`. This is a limited subset of the features necessary to write a source-level debugger. `tproc/decompose` only requires that any procedure that may be passed to `tproc/decompose` must have such information associated with it. We do not need to match arbitrary program states with the original source, a much harder task that source-level debuggers attempt. In addition, only those environments already captured in procedures need to be translated into external data structures, not arbitrary environments. Thus, the task of finding the relevant variables is simpler than in the general case of source-level debugging. In order to implement `tproc/decompose`, the compiler only needs to preserve the information describing the format of environments captured in closures.

We can conclude that the additional complexity required for the implementation of the feature is inexistent for simple implementations, and much smaller than what is required to provide source-level debugging for arbitrary optimizing implementations.

The cost in efficiency to programs that do not use these features is negligible if not zero. The ability to extract expressions and environments from procedures can be implemented with the same mechanisms used to support source-level debugging. The compiler produces object code and debugging information mapping object code locations to source expressions, and suspended states to formal environments. Since we only require this ability of procedures that may be passed to `tproc/decompose`, whose environments must be suspended and collected into the procedure object at run time—and not arbitrary suspended states—the difficulties inherent in inverting arbitrary program states do not appear. In other words, inverting the code is not hard, it just means that the compiler must associate each procedure code location with an expression. Inverting the environment is not difficult either, since the implementation must already collect the environment into a single object (the procedure object) at run time.

As is generally the case, the debugging information need not be in core during execution. It can remain in the files containing object code, and be loaded only on demand. Programs that do not use these features will not be affected.[1]

Finally we can discuss the direct cost of using our operations on translucent procedures. `tproc/make` is a restricted evaluator. It is given a representation of a `lambda` expression, and a representation of an environment, and returns a procedure corresponding to evaluation of the `lambda` expression in the environment. This evaluator can be as uncomplicated as a simple interpreter, or as complicated as a highly-optimizing compiler. Thus the cost of `tproc/make` can vary, with implementations being able to choose between making `tproc/make` itself fast, or making the resulting procedure fast. Of course, ideally, the user/programmer should be able to make the trade-off, rather than the implementor. This can be accomplished by making the native code compiler(s) available to the user/programmer as a transformation between translucent procedures:

(compile-procedure (tproc/make ⟨*expr*⟩ ⟨*env*⟩)) $\mapsto$ ⟨*tproc*⟩

`tproc/decompose` is not inherently slow. Even if the debugging information is maintained out of core, it can be cached after its first use, and collecting the relevant parts of the environment should be a linear process on the size of the environment.

Of course, our canonicalization function and equality predicate may be arbitrarily expensive, even though computable. This does not argue for their elimination, merely

[1]If the implementation does code-generation and constant-folding at run time, the debugging information will have to be generated on the fly as well, increasing the cost of these operations. However, such debugging information can simply consist of the original expression and the values substituted.

for making their use explicit so that programmers can choose whether to use cheaper, more-specific, alternatives, or the general method.

# Chapter 8

# Related and Further Work

## 8.1 Related Work

### 8.1.1 Reflection and Reification

The work presented in this report is most closely related to earlier work on reflection and reification in programming languages and systems [54, 53, 8, 24, 62]. `tproc/decompose` and `tproc/make` are reifying and reflection primitives, respectively, in the terminology used in [24]. However, the focus of this work differs significantly from that of prior work on reflection.

In most of the reflection literature, the goal is to provide a mechanism for metalinguistic abstraction, that is, the ability to extend a programming language within the language itself. This is accomplished by exposing (reifying) components of the internal state of the execution engine. In the present work, arbitrary program states cannot be manipulated; We only manipulate the structure of procedures, and in limited ways. The goal is not metalinguistic abstraction—the extensions are prechosen—but to explore the consequences of violating the opacity of procedures for practical programs. In short, this work, considers a very limited form of reflection, used to extend the range of situations where procedures are a natural and desirable representation for objects; it is an instance of putting limited reflection to practical use.

In spite of the difference in focus, there are similarities between earlier work on reflection, and the present work. Like most Lisp-based work on reflection ([8] excepted), the proposed reification operations provide access to expressions and environments. However, in our case, access is not possible at arbitrary points in the execution of a program, but only from closures, objects that presumably have the expression and environment suspended in non-reified form. Limiting reification to structures that the system must already explicitly represent has important consequences for efficiency, as discussed in Chapter 7.

Finally, reflection work takes two opposite approaches to the reification of procedures. In [53], procedures are completely concrete, exposing all details of their operation, without making any attempt at abstraction. Of course, as we discuss in Chapter 6 this changes the semantics of procedures in fundamental ways. In [24], procedures are completely abstract, with only invocation defined on them; this leaves the semantics of procedures essentially unchanged, but also makes them far less useful, as most of this report argues.

My work pursues a middle ground. Translucent procedures are sufficiently concrete that expressions and environments can be obtained from them, and manipulated by user programs such as the pattern matcher.[1] However, they are sufficiently abstract that the semantics are not compromised in a serious way. Succinctly phrased, translucent procedures compromise the semantics only beyond what our computable (or decidable) mechanical analysis tools can determine.

In part, translucent procedures address an important question common to [53] and [8]. The concrete representation of translucent procedures captures as much of their *intension* as we can determine from a decidable equality predicate. Depending

[1] The representation in [8] is different; a procedure decomposes into a sequence of primitive actions, or tuples representing state transformation operations.

on the *sharpness* of our equality predicate, we can be closer to 3-Lisp [53] or to Brown [24, 62] in the treatment of procedures, with the first falling fully within our framework, and the latter being always beyond reach, but approachable.

The treatment of environments also falls between complete concreteness and complete abstraction, as in [37], but in our case the representation follows directly from our decomposition of procedures.

### 8.1.2 Other Related Work

Pattern matching, and its generalization, *unification,* have been extensively studied both from a practical and a theoretical standpoint. Kevin Knight, in his extensive survey [38], reviews unification, most of the computability results, and its application to logic programming and artificial intelligence. In general, higher-order unification is undecidable [36, 28], although there are interesting cases where it is decidable [44]. By contrast, the decidability of higher-order matching is an open problem [36, 56] although the decidability of certain cases is known [46, 15, 16]. The matcher used here avoids decidability problems by its asymmetry—only the pattern can contain pattern variables, by using finite unfoldment, and by syntactically restricting the values of pattern variables.

The procedural pattern matcher that we use throughout this report, although somewhat ad hoc, shows how higher-order pattern matching can be used to decompose procedures conveniently. Higher-order pattern matchers have previously been used for algebraic specification [32], to abstractly process syntax [43], and in term rewriting systems [23, 45]. Limited higher-order unification is used to make logic programming languages more expressive [48].

There is a great deal of previous work on mechanical equation solvers, and there are even some commercial products [30, 50, 63, 18]. The equation solver presented here is not particularly powerful by the standards of this earlier work. Its salient features are its organization, the choice of procedures as the basic representation, and its ability to encompass both algebraic and numerical methods within the same framework. Methods for finding the roots of equations and for solving linear systems have been studied for centuries. Books on linear algebra [59] and numerical programming [49] describe several such methods and shortcuts that can be used under favorable conditions.

## 8.2 Further Work

The most important item in an agenda for further work is the development of sharper decidable equality predicates. The equality predicate used in this report, which underlies the matcher, is quite limited in that it cannot deal with recursion. The sharper the equality predicate that we use, the more that procedures—even if reifiable—will become like the functions that we want them to denote.

Another important task is to apply the ideas presented here to a more traditional language. Languages without higher-order procedures are not a good test base, because procedures are significantly impaired. However, even languages with higher-order procedures such as Scheme and ML [47], have side effects and mutation, and good translucent representations need to be developed for the visible part of the store.

The procedural matcher can be greatly improved, even if the equality predicate remains fixed. In particular, its efficiency leaves great room for improvement. The matcher, as described in Appendix A, uses two different algorithms. One of them is efficient, but usable only in limited, but common, circumstances. The other, although more general, is very expensive, and makes heavy use of backtracking. Possibilities for the improvement of the matcher are:

- Better treatment of special forms. For example, the current matcher distinguishes between `(begin (begin` $x$ $y$`)` $z$`)` and `(begin` $x$ `(begin` $y$ $z$`))`, and similarly for various combinations of conditionals (`if` expressions). Handling such equivalences will require, among other things, extending it to perform *segment matches* [45].

- Better treatment of primitives. The current matcher does not interpret primitives at all: thus, it distinguishes between `(+` $x$ `1)` and `(-` $x$ `-1)`. A conservative theory of primitive equality would allow more reasoning about function, rather than simply structure. In addition, under certain circumstances, we might want to treat certain compound procedures as primitives, preventing their *expansion* by the matcher.

- The special-case algorithm, although applicable to common cases, is insufficient, while the general algorithm is very expensive. Perhaps a more efficient general algorithm can be found, but even if not, there are probably other large classes of problems where special-purpose matching algorithms can be used to advantage.

- The general algorithm can be improved by using dependency-directed backtracking [55] rather than simple chronological backtracking [20]. Currently, the matcher will attempt the same binding over and over when an inconsistency arises from an earlier binding. Dependency-directed backtracking, although more difficult to code, would probably significantly prune the tree of possibilities.

Finally, the equation solver shown in Chapter 3 is interesting because it elegantly combines algebraic and numeric methods in the same framework. However, there are algebraic techniques that it cannot conveniently use. For example, the system of equations 3.12–3.12 can be solved by using the substitutions 3.9–3.11 in reverse, leading to a linear system (equations 3.1–3.3) that can be solved easily. The system formed by equations 3.9–3.11 can then be solved easily, involving only a quadratic equation. More powerful methods, such as using Gröbner bases [9], are also difficult to integrate in the current framework. These issues should be explored to decide whether solvers using structure similar to the one presented here can be feasible and practical.

# Chapter 9

# Conclusions

In this report, we explore the consequences of violating the traditional opacity of procedures in controlled ways. We show how the ability to inspect procedures makes possible the description of elegant systems in the domains of functional geometry (Chapter 1), equation solvers (Chapter 3), metacircular interpreters and compilers (Chapter 4), and the construction of efficient mathematical subroutines from definitional properties (Chapter 5).

Although the examples presented here are little more than toys, they emphasize the problem with the traditional opaqueness of procedures: *Opaque* procedures are fully abstract objects. The only operation available on opaque procedures is activation, and thus they constitute a perfect representation only for data whose distinguishing property is a *behavior*.

However, there are virtually no objects whose *only* property is a behavior, and opaque procedures hide *all* other aspects. Pictures can not only be drawn and composed, but also transformed and recognized. Equations can be substituted and used to eliminate variables, but also examined to determine the method of attack. Code can be executed, but also pretty-printed, etc.

Even mathematical functions are not well represented by opaque procedures. For example, mathematical functions do not have a *cost* associated with them, but procedures do, yet we can empirically distinguish between a recursive and iterative procedures that compute the same mathematical function by examining resource consumption.

Opaque procedures are too restrictive and abstract to represent objects whose fundamental property is a behavior, but that have other properties as well. In addition, they are not abstract enough to represent objects that are defined purely as maps.

Translucent procedures, introduced in this report as an alternative to traditional procedures, capture both a behavior and a specification. They can be used like ordinary procedures to structure code, and to represent objects with behavior, yet they can be inspected without being invoked. Inspection can be used to discriminate between them and to translate them to alternate representations, when necessary. For example, we can translate translucent procedures representing rational functions into a quotient of polynomials represented as coefficient lists that can be better used to compute polynomial greatest common divisors needed for simplification.

When fundamentally altering the meaning of a long-standing abstraction, one must always carefully consider whether there are unintended effects. Opaque procedures are, together with numbers and arrays, the oldest abstractions in programming languages, and changing their properties can detrimentally affect the semantics and efficiency of a programming language. A naive exposition of the structure of procedures has such drastic consequences.

If we can decompose procedures into the actual expression and environment that give rise to them, they are no longer abstract at all. Their history becomes a more fundamental property than their behavior. Consequently, we have chosen intentionally *blurry* inspection facilities. A translucent procedure can be decomposed into some expression and environment that are equivalent to the actual expression and environment used to create the procedure, but need not be identical to them.

The blurry reification operations hide sufficient information that behavior becomes the primary property again, yet they reveal enough to enable examination and discrimination. We show in Chapter 6that this partly obscured decomposition does not seriously interfere with our expected semantics. We can define our semantics and construct our system to obscure the revealed structure sufficiently that our mechanical program-analysis tools are not affected.

As shown in Chapter 7, the cost of these procedures and their inspection facilities is smaller than, and subsumed by, the cost of implementing source-level debugging facilities in a production implementation.

In summary, translucent procedures, as described in this report, solve the perceived problems of traditional opaque procedures. They are abstract enough that they can be used to represent a behavior, while, simultaneously, they can be examined to discriminate between alternatives, and destructured into their components. The additional abilities added to procedures neither seriously affect the performance of implementations, nor significantly perturb the semantics of the language. Therefore, translucent procedures should be seriously considered when designing or extending programming languages.

# Appendix A

# Implementation details

## A.1 Implementation of TScheme

TScheme is currently implemented as a simple interpreter written in MIT Scheme [31]. Writing an interpreter for a call-by-need dialect of Scheme is a simple exercise [25], left to the reader. The only missing detail is how to make Scheme and TScheme inter-callable.

TScheme procedures are directly invocable from Scheme code, as ordinary procedures, yet they are also data structures that the TScheme interpreter can manipulate directly. Since Scheme only has opaque procedures, if we represented our data as procedures, then it would be difficult to decompose them, while representing them as ordinary data structures (e.g. records or lists) would not make them invocable.[1]

*Entities*, a hybrid data type present in some Scheme implementations, solve the problem cleanly. The precise semantics of entities vary from dialect to dialect, but the fundamental properties are:

- Entities are invocable. They can be invoked and composed as ordinary procedures.

[1] MIT Scheme does not have opaque procedures, but the interface provided is intended for the debugger and is based on the internal data structures of the system, making it cumbersome to use.

- Entities are recognizable. That is, they are distinguished from ordinary procedures by a characteristic predicate.

- Entities have structure. They have two recognized fields that can be extracted.

One of their fields, the *handler*, must contain a procedure that serves as the active component. Invoking the entity is equivalent to invoking the handler with suitable arguments. The other field, the *data*, can contain an arbitrary object.

The version of entities present in MIT Scheme can be described algebraically by the following identities.[2]

$$\begin{aligned}(\mathtt{entity-data}\ (\mathtt{make-entity\ h\ d})) &\equiv \mathtt{d}\\ ((\mathtt{make-entity\ h\ d})\ \mathtt{a1\ldots an}) &\equiv (\mathtt{h}\ (\mathtt{make-entity\ h\ d})\ \mathtt{a1\ldots an})\end{aligned}$$

The following code is a simplification of the code used to construct TScheme procedures:

```
(define (tproc/%make lam env)
  (make-entity (lambda (tproc . args)
                 (tproc/%apply (entity-data tproc)
                               args))
               (cons lam env)))
```

Entities can be approximated in the following way in dialects that do not have them:

```
(define (make-entity h d)
  (letrec ((p
            (lambda args
              (if (or (null? args)
                      (not (null? (cdr args)))
                      (not (eq? (car args) '*GET-DATA*)))
                  (apply h (cons p args))
                  d))))
    p))

(define (entity-data p)
  (p '*GET-DATA*))
```

[2] `entity-data` is called `entity-extra` in MIT Scheme

Of course, implementing a type-specific predicate, and making entity-data safe, is painful and can only be done by maintaining and searching a data structure containing all the entities ever constructed.

Other facilities can be used to implement entities or translucent procedures. Obviously, the implementation on fully reflective languages such as 3-Lisp [54, 53] should be straightforward. Languages and implementations with a meta-object protocol that can be used to modify the behavior of the execution engine [61] also have the means to implement them easily. In addition, other languages, such as C++ [60] have the ability to define invocable objects with structure.

## A.2 Implementation of the Procedural Pattern Matcher

The description of the procedural pattern matcher in chapter 2 is sufficient to follow the examples in the remaining chapters, but inadequate to understand or envision its operation. Although the matcher is only an example of the tools that can be built once procedures become translucent, its frequent and exclusive use in this thesis requires a more detailed description, to avoid leaving the reader with doubts about its feasibility.

The matcher operates by performing a lock-step tree walk on two expressions. The expressions are those returned by `tproc/decompose` when invoked on TScheme procedures. Rather than unfold ($\beta$-substitute) and then compare, the matcher keeps environments binding identifiers in the current expressions to delayed substitutions and other values, in a manner similar to a normal-order evaluator. Substitutions are never performed, but the expressions held in delayed substitutions are compared when needed. This lazy substitution saves the cost of the full substitution when the match can be rejected early, a common occurrence, and avoids having to rename formal parameters to avoid unintended capture.

Since substitution is done implicitly as the matcher walks expressions, circularities (recursion) are also detected during its ordinary operation.

### A.2.1 Data Structures Manipulated by the Matcher

During its operation, the matcher manipulates the following data structures:

1. A *dictionary* binding pattern variables to values. When a new binding for a pattern variable is attempted, the matcher checks the dictionary for compatibility with a previous binding. If there are none, the new binding is added to the dictionary. If there is a previous binding for the pattern variable, the new and previous values are compared for compatibility, and if they are incompatible, the matcher fails. Two bindings are compatible if they are equal (in our sense). If the bindings are compatible, the matcher succeeds and keeps the binding.

2. TScheme *expressions*. These are the expressions being compared subject to the bindings implied by the environments.

3. *Environments* mapping free variables in the expressions to:

    - Values.
    - Delayed substitutions, each consisting of an expression, an environment, and a stack.
    - Special constant tokens used to equate un-substituted formal parameters.

    The initial environments are those returned by `tproc/decompose` on the procedures passed to the matcher as arguments. They map free variables of the `lambda` expression to values. The environments are subsequently extended by binding formal parameters to delayed substitutions when unfolding a combination, and to special tokens when comparing `lambda` expressions of the same arity. The environment corresponding to an expression manipulated by the matcher contains bindings for all of its free variables.

4. *Stacks* recording all the `lambda` expressions unfolded to reach this point. Every time that a combination is unfolded, the matcher checks to make sure that the `lambda` expression used to unfold the combination does not appear (`memq`) in the corresponding stack. If it does, the matcher has detected a circularity, and returns false. Otherwise, it performs the unfoldment by binding the formal parameters to the appropriate delayed substitutions, and records the unfoldment in the stack.

### A.2.2 Comparison Walk in the Matcher

The top-level procedure of the matcher uses `tproc/decompose`[3] on the pattern and instance procedures to provide the initial expressions and environments for the tree walk, and creates an empty dictionary and two empty stacks, one for each expression. It then invokes the main match routine, a procedure that takes seven arguments, namely, a dictionary, two expressions, two environments, and two stacks. When this procedure returns the top-level procedure extracts the relevant parts of the dictionary and returns them if the match is successful.

The main match routine operates by reducing, in turn, the pattern expression, and the instance expression, and then comparing the results. The reduction, which proceeds by replacing variables with their values from the environment, combinations with the bodies of their operators, etc., proceeds until a circularity is detected, or the expression can no longer be reduced. The reduction is similar to normal order reduction until the resulting expression is in head normal form [7], with the difference that substitutions are implied, not carried out, and that a variable is never in head normal form, since it always has a binding in the environment.

Ignoring some special forms (e.g. `if`, `tlet`) which add complexity to the code, but no difficulty, the result of a successful reduction, i.e. a reduction that did not detect

[3]It actually uses `%tproc/decompose`, a *sub-primitive* that does not *force* the delayed bindings in the environment.

circularity, consists of a new environment, a new stack, and one of the following kinds of expression:

- A constant.
- A `lambda` expression.
- A combination whose reduced operator is not a `lambda` expression.

After reducing the pattern expression, the matcher examines the result, and proceeds accordingly:

- If the reduced pattern is a pattern variable (a recognizable constant), it attempts a *simple match*, described later.
- If the reduced pattern is a *combination pattern*, it attempts a *combination match*, described later. A combination pattern is a sequence composed of a pattern variable and a set of combinations, in which each element of the sequence is the reduced operator of the following element in the sequence, and the last element of the sequence is the pattern expression under consideration.

  The following are combination patterns.

  ```
  (((#?FOO x y) (+ z w)) (- x (* y z)))

  (#?BAR x y)
  ```

  The following are not

  ```
  (+ (#?FOO x) (#?BAR y))

  ((cdr (assq #?FOO x)) (#?BAR y))
  ```

- Otherwise the matcher reduces the instance expression, and compares the reduced pattern and instance as follows:

1. If the two expressions are of different types (`if` vs. combination, `lambda` vs. `begin`, etc.), the matcher returns false.

2. If the two expressions are constants, they must be `eqv?` constants.

3. If the two expressions are not `lambda` expressions, the matcher proceeds to recursively compare and match the corresponding components. If any fails the comparison, the matcher fails.

4. If the two expressions are `lambda` expressions, they must have the same number of formal parameters and return the same number of values, otherwise the matcher returns false. If the corresponding numbers are the same, the matcher creates as many special constant tokens as bound variables in either `lambda` expression. The matcher then extends the environments by binding the corresponding variables to the special tokens, and proceeds to compare and match the bodies of the `lambda` expressions in the new environment.

The special tokens introduced in this step are distinguishable from each other and from everything else—each matches only itself. The special tokens will be compared when the variables they are bound to are reduced. These tokens are, conceptually, the new shared bound variable names between the pattern and instance, and they are compared for name equality. As everything else, this lock-step $\alpha$-renaming is done lazily by binding variables in the environment.

For example, if the matcher compares

```
(tlambda (x y)
  (+ x y))
```

with

```
(tlambda (a b)
  (+ a b))
```

it will choose two new tokens, say `[ONE]` and `[TWO]`, and bind `x` to `[ONE]` and `y` to `[TWO]` in the environment used for `(+ x y)`, and it will bind `a` to `[ONE]` and `b` to

[TWO] in the environment used for (+ a b). Later, x and a will both independently reduce to the constant [ONE], and y and b will both independently reduce to the constant [TWO], guaranteeing that the matcher will view them as equal.

If we ignore pattern variables, the matcher is just an equality tester for the condition described in chapter 2. We will see below that the matcher always terminates when it attempts simple or combination bindings, but the reason for termination in other circumstances may not be clear. Termination is guaranteed by the use of the stack. The stack grows monotonically, with lambda expressions being inserted as unfoldments (implied $\beta$-reductions) are performed. Since the program never creates new lambda expressions, and there are only a finite number of lambda expressions in the original program (expression and environment), the stack can only grow to accommodate all these lambda expressions, at which point it will detect circularity and fail.

### A.2.3 Simple Match of Pattern Variables

Simple matches occur when the pattern expression has been reduced to a pattern variable. When this occurs, the matcher substitutes the expression fully, returning false if it detects circularity, and checks that no special constant tokens appear in the output. The result of the match must be an ordinary value. If the fully substituted instance expression contains special tokens introduced when matching lambda expressions, then the instance expression has free variables and cannot yield a valid value, procedure or otherwise; The matcher fails.

For example, assuming that proc is some constant irreducible primitive procedure, the matcher fails when matching

```
(tlambda (x)
  (proc #?FOO))

(tlambda (a)
  (proc (tlambda (b)
          (+ a b))))
```

because the substituted expression corresponding to the instance expression

```
(tlambda (b)
  (+ a b))
```

would contain the special token used to equate `x` and `a`. Of course, pattern variable `#?FOO` could not be bound to a procedure in this case, since this `lambda` expression would have no way to acquire a value for its free variable.

After verifying that the instance does not contain free variables, the matcher binds the pattern variable to the fully substituted instance, unless a previous binding for the pattern variable was not compatible.

### A.2.4 Matching Combination Patterns

A combination match is attempted when the reduced pattern expression is a combination pattern, that is, the curried invocation of a pattern variable. Once the matcher detects this condition, it substitutes the pattern *and* the instance expressions fully. If during this process, it detects a circularity by checking the corresponding stack, it fails. Since both the expression and the pattern have been substituted fully, and there are no cycles, the results can be represented as trees.

The matcher verifies that all the free variables of the instance expression, represented after substitution by special tokens, appear in the pattern expression as well. If there are free variables in the instance that do not appear in the pattern, the matcher fails.

For example,

```
(tlambda (x y) (#?F (* x x)))
```

does not match

```
(tlambda (a b) (+ (* a a) (/ b 3)))
```

because the substituted pattern and instance expressions would be, respectively

```
(#?F (* [ONE] [ONE]))

(+ (* [ONE] [ONE]) (/ [TWO] 3))
```

where the instance contains free variables not found in the pattern.

Except for the final construction of the pattern variable's value, any combination pattern is handled as if it were not curried, and the nesting level of the pattern will be taken into account only when constructing values to be bound.

For example, the treatment of all of

```
(((#?FOO x) y) z)

((#?FOO x) y z)

((#?FOO x y) z)
```

is the same as if they were

```
(#?FOO x y z)
```

except when constructing the `lambda` expression to which `#?FOO` is bound.

Matching combination patterns is often ambiguous [43, 32]. For example,

```
    (#?FOO x x)
```

matches

```
(+ x x)
```

with any of the following bindings:

```
#?FOO ⇒ (tlambda (a b) (+ a b))

#?FOO ⇒ (tlambda (a b) (+ a a))

#?FOO ⇒ (tlambda (b b) (+ b b))
```

The matcher generates only one of the possible matches, but uses backtracking in order to find a satisfactory match if the local choice is rejected elsewhere. A local binding may be rejected when a pattern variable appears more than once. The details of the backtracking mechanism are mundane and not described in the following. The ability to choose and later backtrack to examine an alternate choices is assumed. The backtracking mechanism is chronological [20].

Although, in general, the matcher will use backtracking to find a binding even when pattern variables are not duplicated, there is a common set of cases where ambiguity is limited, and the matching process can be carried out more deterministically and efficiently. The matcher uses two different algorithms for combination matches according to whether the combination pattern is simple or complex.

A *simple* combination pattern satisfies the following conditions:

1. It has no repeated pattern variables.

2. All pattern variables are operators of combination patterns.

3. Every combination pattern that is not the operator of a combination pattern directly appears at top level of another combination pattern, i.e. it is an argument of another pattern variable when all combination patterns are uncurried.

4. No expression that is an argument (when uncurried) of a pattern variable is a subexpression of an argument to a pattern variable except itself.

Any other combination pattern is considered *complex.*

The following are examples of simple combination patterns:

```
(#?FOO (+ x y) z)

((#?FOO x) (#?BAR y))

(((#?FOO (+ x y)) ((#?BAR (- y x)) (+ y x))) (* x y))
```

The following are examples of complex combination patterns.

```
(#?FOO (#?BAR x) (#?BAR y))         ; violates condition 1

(#?FOO (+ x #?BAZ))                 ; violates condition 2

(#?FOO x (+ y (#?BAG z)))           ; violates condition 3
                                    ; because (#?BAG z) is not
                                    ; directly an argument of
                                    ; #?FOO, even after every
                                    ; combination pattern is
                                    ; uncurried.
```

```
(#?FOO x (#?BAR (+ x x)))          ; violates condition 4
                                   ; because X is a subexpression
                                   ; of (+ X X)

(#?FOO (+ x y) (#?BAR (+ x y)))    ; violates condition 4
                                   ; because the leftmost (+ X Y)
                                   ; is a subexpression of the
                                   ; rightmost
```

If there are duplicate pattern variables—the pattern is not simple—each occurrence is considered on its own, and the consistent binding mechanism in the dictionary is used to guarantee that all occurrences are bound to equal values.[4] In the following, we will assume that there are no duplicate pattern variables in either case.

The tree structure of the pattern imposes a tree structure (precedence) on the pattern variables. A pattern variable is considered an ancestor of another if the latter appears in an argument to the former within the pattern. For example, in the pattern `(#?FOO x (#?BAR (+ y (#?BAZ z))))`, `#?FOO` is the ancestor of `#?BAR`, which in turn is the ancestor of `#?BAZ`.

Both algorithms proceed from the innermost (leaf) pattern variables to the root, by binding the inner pattern variables to some pieces of the instance, replacing the pieces of the instance and the corresponding portions of the pattern with new recognizable nodes, and repeating the process until the whole instance and the whole pattern are replaced by new nodes. It is how these nodes are found and replaced that differs between both algorithms. If at any stage no appropriate node is found, or the binding is inconsistent with a previous binding in the dictionary, the matcher fails, possibly backtracking to redo an arbitrary choice.

### Simple Combination Pattern Matching

In order to describe the simple combination pattern matching algorithm, we will examine the algorithm in the context of matching pattern

[4] The equality test is that performed by the matcher in the absence of pattern variables.

```
(tlambda (x y)
  (#?F (#?G (* y y) (* x x))
       (+ x x)))
```

to instance

```
(tlambda (a b)
  (let ((p (+ a a))
        (q (* a a))
        (w (* b b)))
    (+ (+ (+ q w)
          (* p 7))
       (* p
          (sqrt (+ q w))))))
```

where `+`, `*`, and `sqrt` are uninterpreted constant functions.

Initially, the environments contain bindings for `+`, `*`, and `sqrt`. The `tlambda` expressions cannot be reduced further, and they have the same arity and return the same number of values, so the matcher creates special tokens `[ONE]` and `[TWO]` for the arguments and compares the bodies of the `tlambda` expressions in the original environments extended by binding the formal parameters to these tokens.

The pattern

```
(#?F (#?G (* y y) (* x x))
     (+ x x))
```

is reduced in the following environment,

`X` $\mapsto$ `[ONE]`
`Y` $\mapsto$ `[TWO]`
`+` $\mapsto$ $\langle$ *primitive* $+\rangle$
`*` $\mapsto$ $\langle$ *primitive* $*\rangle$

but the result is identical to the input—the pattern is already reduced. The pattern matcher decides that the pattern is a combination pattern, and substitutes the pattern and instance fully, yielding, respectively,[5]

[5]The variables bound to primitives would be replaced by the primitives, but this only makes the expressions larger and no ambiguity arises, so in the following, the names are used instead.

```
(#?F (#?G (* [TWO] [TWO]) (* [ONE] [ONE]))
     (+ [ONE] [ONE]))
```

```
(+ (+ (+ (* [ONE] [ONE]) (* [TWO] [TWO]))
      (* (+ [ONE] [ONE]) 7))
   (* (+ [ONE] [ONE])
      (sqrt (+ (* [ONE] [ONE]) (* [TWO] [TWO])))))
```

At this point the matcher decides that the combination pattern is a simple combination pattern, and verifies that the instance contains no free variables not contained in the pattern.

Up to this point, the process has not been specific to simple patterns. The specific algorithm starts here.

A simple combination can be decomposed into a set of pattern variables and a set of top-level expressions that contain no pattern variables.

For example,

```
((#?FOO (+ x y)) (#?BAR (+ y x)))
```

consists of the pattern variables `#?FOO` and `#?BAR`, and the top-level expressions `(+ x y)` and `(+ y x)`.

Once the matcher decides that it should use the simple combination pattern algorithm, it constructs a directed acyclic graph (DAG) from the instance expression and the top-level expressions of the combination pattern, merging common subexpressions into single nodes.

The top-level expressions of the pattern are

```
(* [ONE] [ONE])
(* [TWO] [TWO])
(+ [ONE] [ONE])
```

and the DAG constructed appears in fig. A-1.

The matcher then picks variable names for each top-level expression in the pattern (`m`, `n`, and `o`), and replaces the corresponding nodes in the DAG with nodes corresponding to these variables ($Nm$, $Nn$, and $No$). It also rewrites the pattern to reflect the substitution of the top-level expressions.

The rewritten pattern is

```
(#?F (#?G m n) o)
```

and the edited graph appears in fig. A-2, corresponding to the following expression.

```
(let (($ (+ n m)))
  (+ (+ $ (* o 7))
     (* o (sqrt $))))
```

After rewriting the pattern, the dominators for all nodes in the DAG are computed. A node $a$ dominates a node $b$ if all paths from the root to $b$ pass through $a$. Dominators can be computed using the standard algorithm [2].[6]

After computing the dominators, the matcher proceeds to bind pattern variables by processing combination patterns from the innermost to the outermost. If there are multiple combination patterns at the same innermost nesting level, they can be processed in any order, since the subgraphs will not overlap because of defining property 4 of simple combination patterns—we may as well choose the leftmost.

In our example, the innermost combination pattern is

```
(#?G m n)
```

The common dominators of Nm and Nn are found. If there are none, the matcher fails.[7] In the graph in fig. A-2, the common dominators are the node labeled *N$* and the root node. Of these common dominators, those that are ancestors of variable nodes not present in the combination pattern are eliminated. In our example, the root node is eliminated because it is also an ancestor of *No*, which does not appear in the pattern under consideration. This leaves *N$* as the only remaining node. In general this may leave multiple candidates, and the matcher arbitrarily chooses the one closest to the root, examining the other choices only when the matcher backtracks.

[6]The standard algorithm handles graphs with cycles, but our graph has no cycles, and dominators can be computed more easily: the dominators of a node consist of the node and the intersection of the dominators of the direct ancestors. The root is only dominated by itself, and a DAG can be walked in an order where all ancestors of a node are processed before the node itself is.

[7]Only those nodes actually appearing in the instance are considered. The rest are ignored.

The matcher then constructs a value for pattern variable `#?G` by converting the chosen node (*N$*) to an expression, and binding the names of the nodes appearing in the pattern, currying them appropriately. The candidate binding in our case is

```
#?G ⇛ (tlambda (m n) (+ n m))
```

Assuming that the binding can be inserted in the dictionary, the matcher picks a new variable name (`l`) and a variable node (*Nl*) to replace the chosen node (*N$*), edits the graph appropriately, and rewrites the pattern as

```
(#?F l o)
```

The edited graph is shown in fig. A-3, corresponding to the following expression.

```
(+ (+ l (* o 7))
   (* o (sqrt l)))
```

The process is repeated with the new innermost-leftmost combination pattern, which happens to be the top-level combination pattern. The single common dominator of *Nl* and *No* is the root node, and it is not the ancestor of any variable node not present in the pattern under consideration. The binding for `#?F` is constructed and inserted in the dictionary. The binding is:

```
#?F ⇛ (tlambda (l o)
         (+ (+ l (* o 7))
            (* o (sqrt l))))
```

A new variable, `w`, is generated, and a corresponding node, *Nw* created. The chosen node is replaced in the DAG with *Nw*, and the combination pattern is rewritten as

```
w
```

There are no combination patterns left, the matcher verifies that the pattern corresponds to the single node remaining in the DAG, and returns.

### Complex Combination Pattern Matching

The complex combination pattern matching is modeled after the algorithm for simple combination patterns, but uses backtracking heavily. Instead of using a DAG, which was valid because the subexpressions of the pattern could be uniquely identified in the tree, the complex combination pattern matcher uses trees, and backtracks to explore sets of equivalent nodes that would have been merged into a single node in the DAG.

The instance expression is transformed into a tree, and for each node, we compute the set of ancestor nodes and set of free variable descendants, i.e. the set of nodes corresponding to special tokens that can be reached from the node in question.

The pattern is examined from the innermost to the outermost pattern variable.

If the innermost pattern variable is the operator of a combination pattern, its value must be a constant or a `lambda` expression with no free variables. All such possibilities are explored by backtracking.

If the innermost pattern variable is not the operator of a combination pattern, the operands of the combination pattern do not contain pattern variables (otherwise they would not be the innermost). First we find each operand in the instance by computing the set of free variables of the operand and comparing it with the nodes in the instance tree that have the same set of free variables. There may be multiple nodes in the tree that match the operand. When there are multiple nodes, we explore all possibilities by backtracking, and we also consider the case where no node is found for an operand.

After obtaining a set of nodes matching the operands of the combination pattern, we ignore those operands for which we have found no matching nodes, and compute the common set of ancestors of the rest.[8] From the common set of ancestors, which always includes the root of the tree, we eliminate those nodes that contain free variables not found in any of the operands under consideration.

From the remaining nodes, we arbitrarily choose a node, using backtracking to

[8] The set may be empty—the binding is under-constrained.

explore other choices, find other nodes equivalent to it in the set, and from this subset arbitrarily choose a subset of nodes to replace in the instance. Alternate subsets and choices for the node are explored by backtracking.

Once we have chosen a set of equivalent nodes to replace, we construct the binding for the pattern variable as in the simple combination pattern matching algorithm, bind it in the dictionary, generate a new free variable (special token) and a corresponding node, replace the chosen nodes with the new node, and the combination pattern in the overall pattern with the new free variable. We then recompute the ancestor and free variable sets for each affected node in the instance, i.e. the ancestors of the nodes chosen. If the updated instance contains free variables no longer found in the pattern, the matcher fails.

If the matcher succeeds, we repeat the process with the next innermost pattern variable.

At the end of the process, we verify that a single variable has replaced the pattern, and that the corresponding node has replaced the whole instance. Otherwise the matcher fails.

### A.2.5 Under-constrained Values and Consistent Bindings

Consider the following pattern and instance,

```
(tlambda (x) (#?FOO (+ x x) (#?BAR (* x x))))
```

```
(tlambda (x) (* (+ x x) 3))
```

which match with the following bindings:

`#?FOO` $\Leftarrow$ `(lambda (y z) (* y 3))`

`#?BAR` $\Leftarrow$ `(lambda (q)` $\langle$*anything*$\rangle$`)`

The binding for `#?BAR` is considered under-constrained and inserted as such in the dictionary. When a pattern variable is bound in the dictionary, old values are compared for consistency. If neither value is under-constrained, they must be equal.

Otherwise their signatures must match, and the more specific, if either, is kept in the dictionary.

Under-constrained values can arise from simple combination patterns such as the above, but typically arise from choosing empty sets of nodes to replace in the complex combination matcher. Most such choices are soon rejected because some free variable will have been removed from the pattern but not from the instance.

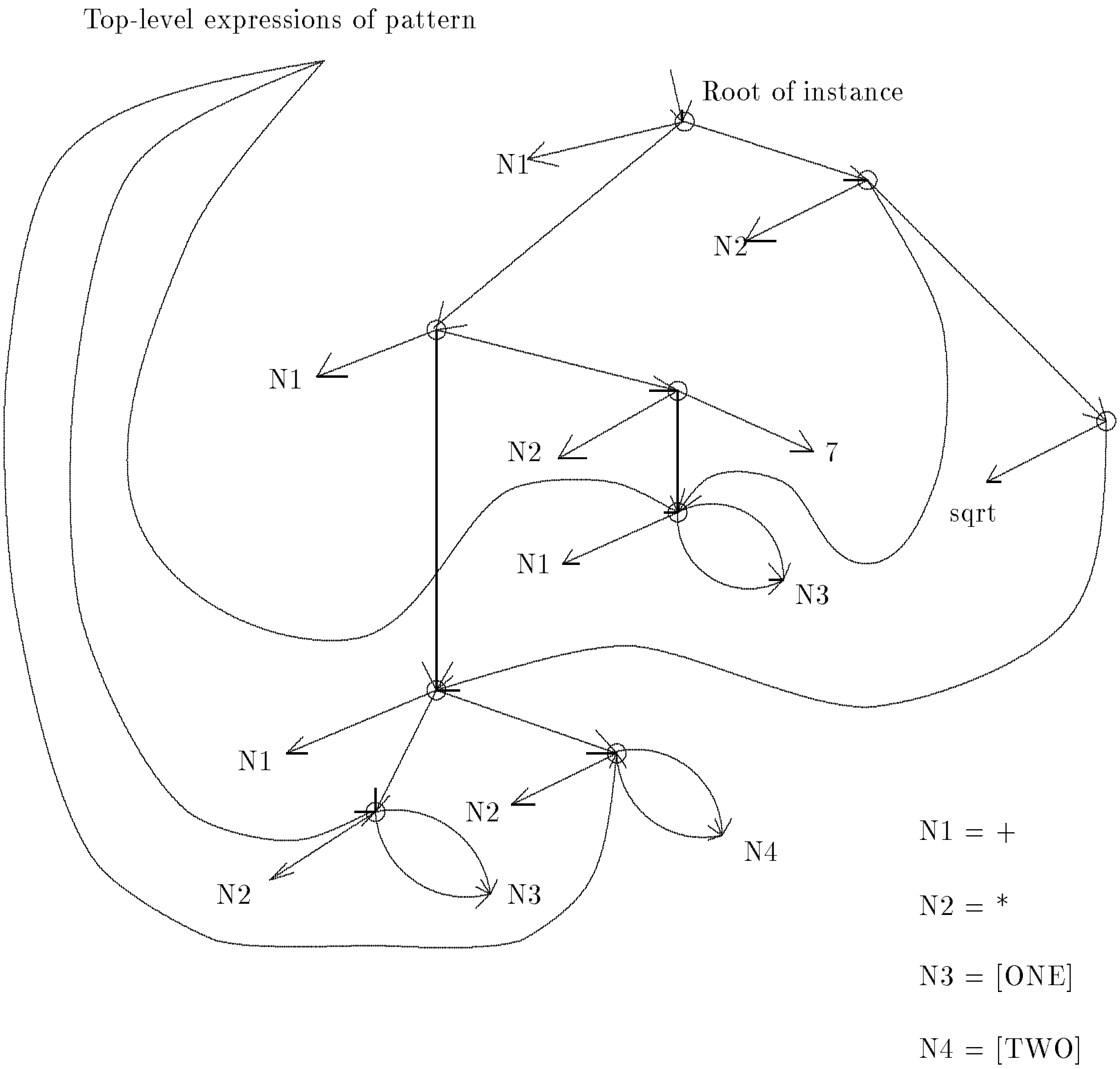


Figure A-1: Initial DAG

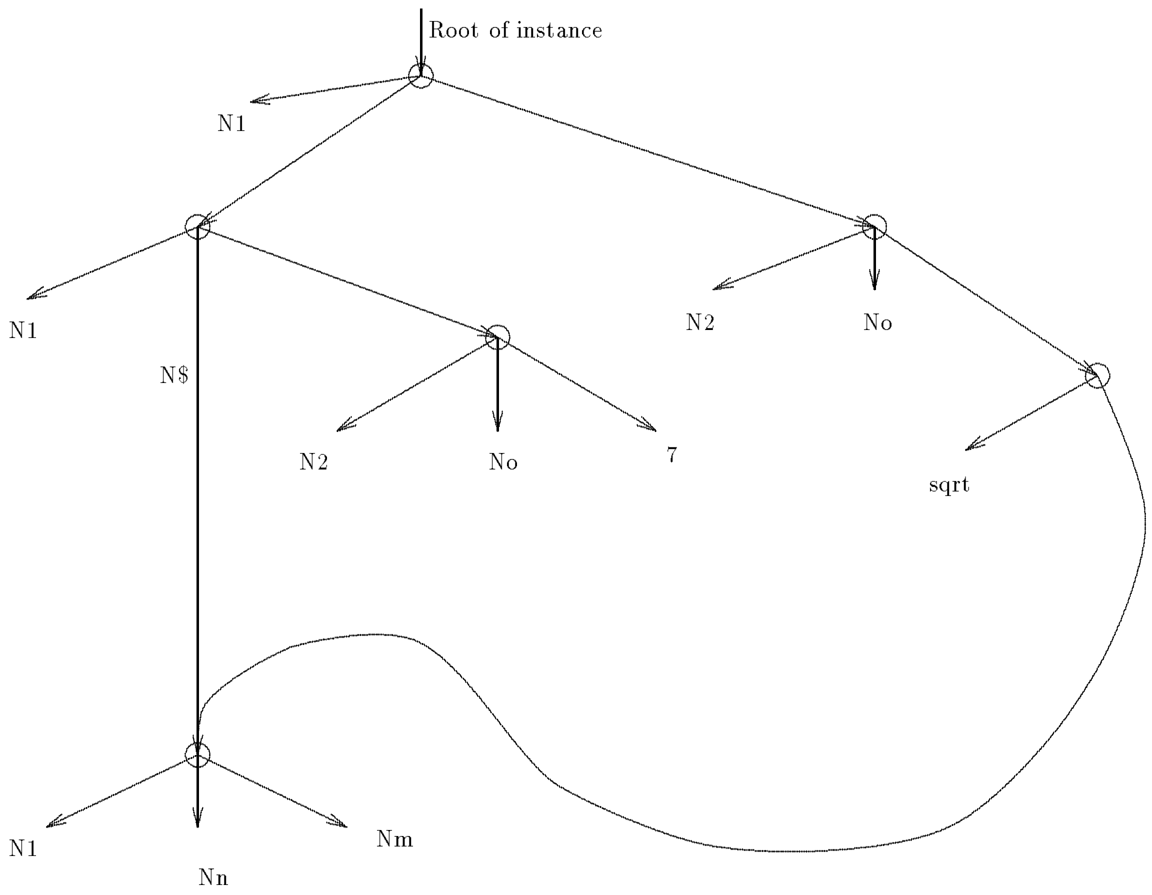


Figure A-2: DAG after replacing expressions with new nodes

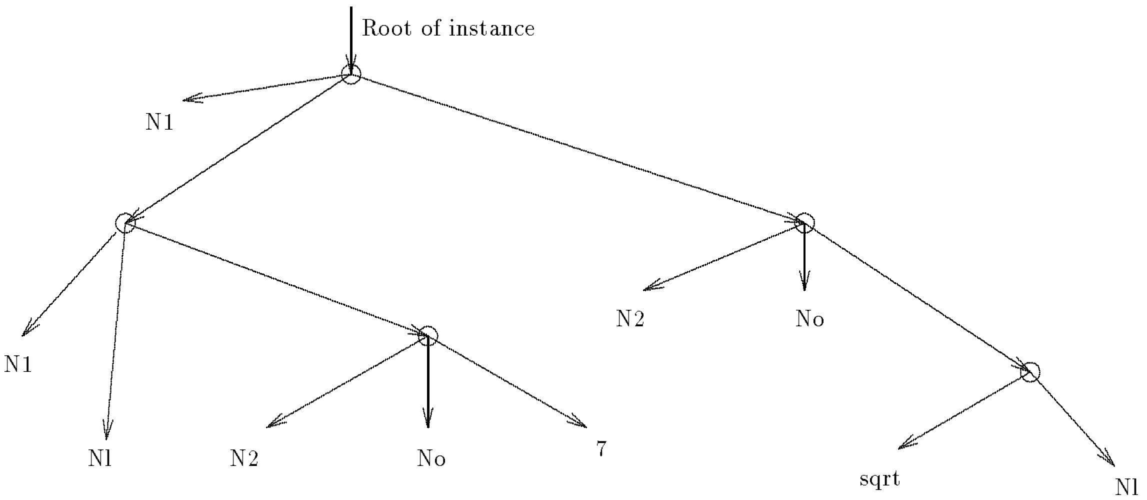


Figure A-3: DAG after binding `#?G`

# Appendix B

# Denotational Semantics

The following is a denotational semantics for a call-by-value dialect of Scheme with translucent procedures. Making it call-by-need, like TScheme, would be cumbersome but not elucidating.

Two versions of the semantics are provided. One is written in Scheme. The other in the traditional syntax.

This semantics does not model `tproc?`, `tproc/arity`, `tproc/nvalues`, `tuple`, or `tlet`. Describing their meaning is not difficult (only cumbersome), but detracts from the legibility of the essential parts. In addition, like TScheme, the language below only has a `lambda` operator. All user-defined procedures are translucent. Finally, no syntactic abstraction facilities (i.e. macros) are provided.

## B.1 Semantics in Scheme notation

```
#| % -*- Scheme -*-

(I) Ide                                 identifiers

(E) Exp ->  K | I | (E0 E*)             expressions
          | (lambda (I*) E0)
          | (if E0 E1 E2)
```

```
Bin = Ide x V                           reified bindings

REnv = Bin*                             reified environments

Clo = Exp x REnv                        closures (reified procedures)

F = V* -> K                             functions

P = Ide x F                             primitive procedures

T = Clo x F                             compound (translucent) procedures

(v) V =                                 expressible values
      N                                 natural numbers
    + B                                 booleans
    + Ide
    + Exp
    + Bin
    + REnv
    + Clo
    + P
    + T

(r) U = Ide -> V                        environments

(q) K = V -> V                          expression continuations

K* = V* -> V

EEval: Exp -> U -> K

EEval*: Exp* -> U -> K*

|#

;; EEval is called Meaning in the text.

(define (EEval E)
  (cond ((IsConst? E)
         (let ((K E))
           ;; (EEval K)
           (lambda (r)
             (return (Const->Value K)))))
        ((IsIde? E)
         (let ((I E))
```

```
           ;; (EEval I)
           (lambda (r)
             (let ((v (Lookup r I)))
               (if (Undefined? v)
                   (abort "Undefined variable")
                   (return v))))))
        ((IsCall? E)
         (let ((E0 (Call/Operator E))
               (E* (Call/Operands E)))
           ;; (EEval (E0 E*))
           (lambda (r)
             (lambda (q)
               (((EEval* (concat E0 E*))
                 r)
                (lambda (v*)
                  (((EApply (hd v*))
                    (tl v*))
                   q)))))))
        ((IsLambda? E)
         (let ((I* (Lambda/Formals E))
               (E0 (Lambda/Body E)))
           ;; (EEval (lambda (I*) E0))
           (lambda (r)
             (Enclose E r I* E0))))
        ((IsIf? E)
         (let ((E0 (If/Predicate E))
               (E1 (If/Consequent E))
               (E2 (If/Alternative E)))
           ;; (EEval (if E0 E1 E2))
           (lambda (r)
             (lambda (q)
               (((EEval E0) r)
                (lambda (p)
                  (if (True? p)
                      (((EEval E1) r) q)
                      (((EEval E2) r) q))))))))
        (else
         (s-error E))))

(define (EEval* E*)
  (lambda (r)
    (if (IsNull? E*)
        (return* empty)
        (lambda (q)
          (((EEval (hd E*))
```

```
           r)
          (lambda (v)
            (((EEval* (tl E*))
              r)
             (lambda (v*)
               ((return* (concat v v*))
                q)))))))))

(define (Enclose E r I* E0)
  (return
   (Make-Procedure
    (Canonicalize-Closure (Make-Closure E r))
    (lambda (v*)
      (if (not (= (num I*) (num v*)))
          (abort "Wrong number of arguments")
          (lambda (q+)
            (((EEval E0)
              (Extend r I* v*))
             q+)))))))

(define (Eapply p)
  (cond ((IsProcedure? p)
         (Procedure->Function p))
        ((IsPrimitive? p)
         (Primitive->Function p))
        (else
         (abort "Bad procedure"))))

(define tproc?
  (lambda (v*)
    (cond ((not (= (num v*) 1))
           (abort "Wrong number of arguments"))
          (else
           (return (IsProcedure? (hd v*)))))))

(define tproc/make
  (lambda (v*)
    (cond ((not (= (num v*) 2))
           (abort "Wrong number of arguments"))
          ((not (IsLambda? (hd v*)))
           (abort "Bad expression"))
          ((not (IsRenv? (hd (tl v*))))
           (abort "Bad environment"))
          (else
           (let ((E (hd v*))
```

```
                 (r (REnv->Environment (hd (tl v*)))))
             (Enclose E
                      r
                      (Lambda/Formals E)
                      (Lambda/Body E)))))))

(define tproc/decompose
  (lambda (v*)
    (cond ((not (= (num v*) 1))
           (abort "Wrong number of arguments"))
          ((not (IsProcedure? (hd v*)))
           (abort "Wrong type argument"))
          (else
           (return (Procedure->Closure (hd v*)))))))

(define ->expr
  (lambda (v*)
    (cond ((not (= (num v*) 1))
           (abort "Wrong number of arguments"))
          ((not (IsClosure? (hd v*)))
           (abort "Wrong type argument"))
          (else
           (return (Closure->Expression (hd v*)))))))

(define ->env
  (lambda (v*)
    (cond ((not (= (num v*) 1))
           (abort "Wrong number of arguments"))
          ((not (IsClosure? (hd v*)))
           (abort "Wrong type argument"))
          (else
           (return (Closure->REnv (hd v*)))))))

;; Continuations

(define (return v)
  (lambda (k)
    (k v)))

(define (return* v*)
  (lambda (k)
    (k v*)))

;; Environments
```

```
(define (Make-Empty-Environment)
  (lambda (I)
    *undefined*))

(define (Lookup r I)
  (r I))

(define (Extend r I* v*)
  (if (IsNull? I*)
      r
      (Extend (Extend-1 r (hd I*) (hd v*))
              (tl I*)
              (tl v*))))

(define (Extend-1 r I v)
  (lambda (I+)
    (if (Ide=? I I+)
        v
        (r I+))))

;; Translucent procedures

(define *procedure-tag*
  (string->symbol "#Procedure"))

(define (Make-Procedure c f)
  (cons *procedure-tag*
        (list c f)))

(define (IsProcedure? v)
  (and (pair? v)
       (eq? (car v) *procedure-tag*)))

(define (Procedure->Function p)
  (cadr (cdr p)))

(define (Procedure->Closure p)
  (car (cdr p)))

;; Concrete procedure representation

(define *closure-tag*
  (string->symbol "#Closure"))

(define (Make-Closure E r)
```

```
  (cons *closure-tag*
        (list E (Environment->REnv r (FreeVars E)))))

(define (IsClosure? v)
  (and (pair? v)
       (eq? (car v) *closure-tag*)))

(define (Closure->Expression v)
  (car (cdr v)))

(define (Closure->REnv v)
  (cadr (cdr v)))

(define (Canonicalize-Closure Clo)
  ;; *** Here is the magic ***
  Clo)

(define (Environment->REnv r I*)
  (if (IsNull? I*)
      empty
      (concat (Make-Binding (hd I*)
                            (Lookup r (hd I*)))
              (Environment->REnv
               r
               (tl I*)))))

(define *binding-tag*
  (string->symbol "#Binding"))

(define (Make-Binding I v)
  (cons *binding-tag*
        (cons I v)))

(define (IsBinding? v)
  (and (pair? v)
       (eq? *binding-tag*
            (car v))))

(define (Binding/Identifier b)
  (car (cdr b)))

(define (Binding/Value b)
  (cdr (cdr b)))

(define (IsRenv? v)
```

```
  (or (IsNull? v)
      (and (IsBinding? (hd v))
           (IsRenv? (tl v)))))

(define (REnv->Environment v)
  (if (IsNull? v)
      (Make-Empty-Environment)
      (Extend-1 (REnv->Environment (tl v))
                (Binding/Identifier (hd v))
                (Binding/Value (hd v)))))



(define (IsExpression? E)
  (or (IsConst? E)
      (IsIde? E)
      (IsCall? E)
      (IsLambda? E)
      (IsIf? E)))

(define (FreeVars E)
  (cond ((IsConst? E)
         (let ((K E))
           ;; (FreeVars K)
           empty))
        ((IsIde? E)
         (let ((I E))
           ;; (FreeVars I)
           (concat I empty)))
        ((IsCall? E)
         (let ((E0 (Call/Operator E))
               (E* (Call/Operands E)))
           ;; (FreeVars (E0 E*))
           (Ide-union (FreeVars E0)
                      (FreeVars* E*))))
        ((IsLambda? E)
         (let ((I* (Lambda/Formals E))
               (E0 (Lambda/Body E)))
           ;; (FreeVars (lambda (I*) E0))
           (Ide-difference (FreeVars E0)
                           I*)))
        ((IsIf? E)
         (let ((E0 (If/Predicate E))
               (E1 (If/Consequent E))
               (E2 (If/Alternative E)))
```

```
           ;; (Freevars (if E0 E1 E2))
           (Ide-union (FreeVars E0)
                      (Ide-union (FreeVars E1)
                                 (FreeVars E2)))))
        (else
         (s-error E))))

(define (FreeVars* E*)
  (if (IsNull? E*)
      empty
      (Ide-union (FreeVars (hd E*))
                 (FreeVars* (tl E*)))))

(define (Ide-union I1* I2*)
  (cond ((IsNull? I1*)
         I2*)
        ((Ide-member? (hd I1*) I2*)
         (Ide-union (tl I1*) I2*))
        (else
         (concat (hd I1*)
                 (Ide-union (tl I1*) I2*)))))

(define (Ide-difference I1* I2*)
  (cond ((IsNull? I1*)
         empty)
        ((Ide-member? (hd I1*) I2*)
         (Ide-difference (tl I1*) I2*))
        (else
         (concat (hd I1*)
                 (Ide-difference (tl I1*) I2*)))))

(define (Ide-member? I I*)
  (cond ((IsNull? I*)
         false)
        ((Ide=? I (hd I*))
         true)
        (else
         (Ide-member? I (tl I*)))))

;; Values

(define (True? v)
  (not (eq? v #f)))

(define *undefined*
```

```
  (string->symbol "#undefined"))

(define (Undefined? v)
  (eq? v *undefined*))

(define *primitive-tag*
  (string->symbol "#Primitive"))

(define (Make-Primitive name p)
  (cons *primitive-tag* (list p name)))

(define (IsPrimitive? v)
  (and (pair? v)
       (eq? (car v) *primitive-tag*)))

(define (Primitive->Function v)
  (car (cdr v)))

(define (Primitive->Name v)
  (cadr (cdr v)))

(define (Make-Binary-Numeric-Primitive name p)
  (Make-Primitive
   name
   (lambda (v*)
     (if (not (= (num v*) 2))
         (abort "Wrong number of arguments")
         (if (or (not (number? (hd v*)))
                 (not (number? (hd (tl v*)))))
             (abort "Wrong type of arguments")
             (return (p (hd v*) (hd (tl v*)))))))))

(define (Make-Unary-Primitive name p)
  (Make-Primitive
   name
   (lambda (v*)
     (if (not (= (num v*) 1))
         (abort "Wrong number of arguments")
         (return (p (hd v*)))))))

(define (Make-Binary-Primitive name p)
  (Make-Primitive
   name
   (lambda (v*)
     (if (not (= (num v*) 2))
```

```
         (abort "Wrong number of arguments")
         (return (p (hd v*) (hd (tl v*))))))))

(define (Make-Special-Primitive name p)
  (Make-Primitive
   name
   p))

;; Syntax

(define (IsConst? E)
  (or (number? E)
      (eq? E #f)
      (eq? E #t)))

(define (Const->Value K)
  K)

(define (IsIde? E)
  (symbol? E))

(define (Ide=? I1 I2)
  (eq? I1 I2))

(define (IsCall? E)
  (and (pair? E)
       (not (memq (car E) '(lambda if)))))

(define (Call/Operator E)
  (car E))

(define (Call/Operands E)
  (cdr E))

(define (IsLambda? E)
  (and (pair? E)
       (eq? (car E) 'lambda)))

(define (Lambda/Formals E)
  (cadr E))

(define (Lambda/Body E)
  (caddr E))

(define (IsIf? E)
```

```
  (and (pair? E)
       (eq? (car E) 'if)))

(define (If/Predicate E)
  (cadr E))

(define (If/Consequent E)
  (caddr E))

(define (If/Alternative E)
  (cadddr E))

;; Random

(define (IsNull? x*)
  (null? x*))

(define (hd x*)
  (car x*))

(define (tl x*)
  (cdr x*))

(define (num x*)
  (length x*))

(define (concat x x*)
  (cons x x*))

(define empty
  '())

;; Top-Level

(define (Make-Empty-Environment)
  (lambda (I)
    *undefined*))

(define (Make-Initial-Environment)
  (let ((names-a      '(+ - * / =))
        (values-a (list + - * / =))
        (names1      '(
                       Primitive?
                       Procedure?
                       Closure?
```

```
                Environment?
                Binding?
                Binding/Identifier
                Binding/Value

                hd
                tl

                Const?
                Ide?
                Lambda?
                Call?

                #|
                Expression?
                Lambda/Formals
                Lambda/Body
                Make-Lambda
                <More such here>
                |#
                ))
  (values1 (list IsPrimitive?
                IsProcedure?
                IsClosure?

                IsRenv?
                IsBinding?
                Binding/Identifier
                Binding/Value

                hd
                tl

                IsConst?
                IsIde?
                IsLambda?
                IsCall?

                #|
                IsExpression?
                Lambda/Formals
                Lambda/Body
                Make-Lambda
                <More such here>
```

```
                      |#
                      ))

        (names2      '(Ide=?))
        (values2 (list Ide=?))

        (names-x      '(
                        tproc?
                        tproc/make
                        tproc/decompose
                        ->expr
                        ->env))
        (values-x (list tproc?
                        tproc/make
                        tproc/decompose
                        ->expr
                        ->env)))
    (Extend
     (Extend
      (Extend
       (Extend (Make-Empty-Environment)
               names-a
               (map Make-Binary-Numeric-Primitive
                    names-a
                    values-a))
       names1
       (map Make-Unary-Primitive
            names1
            values1))
      names2
      (map Make-Binary-Primitive
           names2
           values2))
     names-x
     (map Make-Special-Primitive
          names-x
          values-x))))

(define (Make-Initial-Continuation)
  (lambda (v)
    v))

(define (Go Exp)
  (((EEval Exp)
    (Make-Initial-Environment))
```

```
   (Make-Initial-Continuation)))

(define (abort string)
  (lambda (q)                          ; continuation
    (error string)))

(define (s-error E)
  (error "Illegal expression" E))
```

## B.2 Semantics in Traditional notation

### B.2.1 Abstract Syntax

| | | | |
|---|---|---|---|
| K | $\in$ | Con | constants |
| I | $\in$ | Ide | identifiers |
| E | $\in$ | Exp | expressions |

### B.2.2 Domain Equations

| | | | | | |
|---|---|---|---|---|---|
| $\beta$ | $\in$ | Bin | = | Ide $\times$ V | reified bindings |
| | | REnv | = | Bin$^*$ | reified environments |
| | | Clo | = | Exp $\times$ REnv | closures (reified procedures) |
| | | F | = | V$^*$ $\to$ K $\to$ V | functions |
| | | P | = | Ide $\times$ F | primitive procedures |
| | | T | = | Clo $\times$ F | compound (translucent) procedures |
| | | | | | |
| $\nu$ | $\in$ | V | = | | expressible values |
| | | | | N | natural numbers |
| | | | | + B | booleans |
| | | | | + Ide | |
| | | | | + Exp | |
| | | | | + Bin | |
| | | | | + REnv | |
| | | | | + Clo | |
| | | | | + P | |
| | | | | + T | |
| | | | | + $\{Undefined\}$ | undefined value |
| | | | | | |
| $\rho$ | $\in$ | U | = | Ide $\to$ V | environments |
| $\kappa$ | $\in$ | K | = | V $\to$ V | continuations |
| | | K$^*$ | = | V$^*$ $\to$ V | |

### B.2.3 Semantic Functions

$\mathcal{K} : \mathrm{Con} \to \mathrm{V}$
$\mathcal{E} : \mathrm{Exp} \to \mathrm{U} \to \mathrm{K} \to \mathrm{V}$
$\mathcal{E}^* : \mathrm{Exp}^* \to \mathrm{U} \to \mathrm{K} \to \mathrm{V}$
$\mathcal{E}_\lambda : \mathrm{Exp} \to \mathrm{U} \to \mathrm{K} \to \mathrm{V}$
$\mathcal{F} : \mathrm{Exp} \to \mathrm{Ide}^*$
$\mathcal{F}^* : \mathrm{Exp} \to \mathrm{Ide}^*$

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho.\ \mathit{return}\ (\mathcal{K}[\![\mathrm{K}]\!])$$

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho.\ & (\rho\mathrm{I} = \mathit{Undefined}) \rightarrow \mathit{wrong},\\ & \mathit{return}\ (\rho\mathrm{I})\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![(\mathrm{E}_0\ \ \mathrm{E}^*)]\!] = \lambda\rho\kappa.\ \mathcal{E}^*(&\langle\mathrm{E}_0\rangle\ \S\ \mathrm{E}^*)\\ &\rho\\ &(\lambda\nu^*.\ \mathit{apply}\ (\nu^* \downarrow 1)(\nu^* \dagger 1)\kappa)\end{aligned}$$

$$\mathcal{E}[\![(\texttt{lambda}\ \ (\mathrm{I}^*)\ \ \mathrm{E}_0)]\!] = \lambda\rho.\ \mathit{enclose}\ \rho\ \mathrm{I}^*\ \mathrm{E}_0$$

$$\begin{aligned}\mathcal{E}[\![(\texttt{if}\ \ \mathrm{E}_0\ \ \mathrm{E}_1\ \ \mathrm{E}_2)]\!] = \lambda\rho\kappa.\ \mathcal{E}&[\![\mathrm{E}_0]\!]\\ &\rho\\ &\lambda\nu.\ (\mathit{truish}\ \nu) \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\ &\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa\end{aligned}$$

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

$$\begin{aligned}\mathcal{E}^*[\![\mathrm{E}_0\ \ \mathrm{E}^*]\!] = \lambda\rho\kappa.\ \mathcal{E}&[\![\mathrm{E}_0]\!]\\ &\rho\\ &(\lambda\nu_0.\ \mathcal{E}^*[\![\mathrm{E}^*]\!]\\ &\qquad\qquad \rho\\ &\qquad\qquad (\lambda\nu^*.\ \kappa\ (\langle\nu_0\rangle\ \S\ \nu^*)))\end{aligned}$$

$$\mathit{enclose} : \mathrm{U} \rightarrow \mathrm{Ide}^* \rightarrow \mathrm{Exp} \rightarrow \mathrm{K} \rightarrow \mathrm{V}$$

$$\begin{aligned}\mathit{enclose} = \lambda\rho\mathrm{I}^*\mathrm{E}.&\\ &\mathit{return}\ \langle(\mathit{canon}\ (\mathit{makeclo}\ (\texttt{lambda}\ \ (\mathrm{I}^*)\ \ \mathrm{E})\ \rho))\\ &\qquad\qquad |(\lambda\nu^*\kappa'.\ (\#\mathrm{I}^* \neq \#\nu^*) \rightarrow \mathit{wrong},\\ &\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}]\!](\mathit{extend}\ \rho\ \mathrm{I}^*\ \nu^*)\kappa')\rangle\\ &\qquad\qquad \mathit{in}\ \mathrm{V}\end{aligned}$$

$$\mathit{canon} : \mathrm{Clo} \rightarrow \mathrm{Clo} \qquad\qquad [\text{Unspecified}]$$

$$\mathit{apply} : \mathrm{V} \rightarrow \mathrm{V}^* \rightarrow \mathrm{K} \rightarrow \mathrm{V}$$

$$\begin{aligned}\mathit{apply} = \lambda\nu\nu^*\kappa.\ & (\nu \in \mathrm{T}) \rightarrow (\nu|\mathrm{P} \downarrow 2)\nu^*\kappa,\\ & (\nu \in \mathrm{P}) \rightarrow (\nu|\mathrm{M} \downarrow 2)\nu^*\kappa,\\ & \mathit{wrong}\end{aligned}$$

$extend : \mathrm{U} \to \mathrm{Ide}^* \to \mathrm{V}^* \to \mathrm{U}$

$$\begin{aligned} extend = \lambda\rho \mathrm{I}^* \nu^*.\ & (\#\mathrm{I}^* = 0) \to \rho, \\ & extend\,(\rho[(\nu^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)]) \\ & \qquad\quad (\mathrm{I}^* \dagger 1) \\ & \qquad\quad (\nu^* \dagger 1) \end{aligned}$$

**Auxiliary Functions**

$truish : \mathrm{V} \to \mathrm{B}$

$$\begin{aligned} truish = \lambda\nu.\ & (\nu \notin \mathrm{B}) \to true, \\ & (\nu|\mathrm{B} = false) \to false, \\ & true \end{aligned}$$

$return : \mathrm{V} \to \mathrm{K} \to \mathrm{V}$

$return = \lambda\nu\kappa.\ \kappa\nu$

$makeclo : \mathrm{Exp} \to \mathrm{U} \to \mathrm{Clo}$

$makeclo = \lambda \mathrm{E}\rho.\ \langle \mathrm{E}|(reifyenv\ \rho\ \mathcal{F}[\![\mathrm{E}]\!])\rangle\ in\ \mathrm{Clo}$

$reifyenv : \mathrm{U} \to \mathrm{Ide}^* \to \mathrm{REnv}$

$$\begin{aligned} reifyenv = \lambda\rho \mathrm{I}^*.\ & (\#\mathrm{I}^* = 0) \to \langle\ \rangle, \\ & (makebin\ \rho\ (\mathrm{I}^* \downarrow 1))\ \S\ (reifyenv\ \rho\ (\mathrm{I}^* \dagger 1)) \end{aligned}$$

$makebin : \mathrm{U} \to \mathrm{Ide} \to \mathrm{Bin}$

$makebin = \lambda\rho \mathrm{I}.\ \langle \mathrm{I}|(\rho\ \mathrm{I})\rangle\ in\ \mathrm{Bin}$

$\mathcal{F}[\![\mathrm{K}]\!] = \langle\ \rangle$

$\mathcal{F}[\![\mathrm{I}]\!] = \langle \mathrm{I} \rangle$

$\mathcal{F}[\![(\mathrm{E}_0\ \ \mathrm{E}^*)]\!] = \mathcal{F}^*[\![\langle \mathrm{E}_0\rangle\ \S\ \mathrm{E}^*]\!]$

$\mathcal{F}[\![(\texttt{lambda}\ \ (\mathrm{I}^*)\ \ \mathrm{E}_0)]\!] = seqdif\ (\mathcal{F}[\![\mathrm{E}_0]\!])\ \mathrm{I}^*$

$$\mathcal{F}[\![(\texttt{if}\ \ \mathrm{E}_0\ \ \mathrm{E}_1\ \ \mathrm{E}_2)]\!] = \mathcal{F}^*[\![\langle \mathrm{E}_0\rangle \ \S\ \langle \mathrm{E}_1\rangle \ \S\ \langle \mathrm{E}_2\rangle]\!]$$

$$\mathcal{F}^*[\![\ \ ]\!] = \langle\ \rangle$$

$$\mathcal{F}^*[\![\mathrm{E}_0\ \ \mathrm{E}^*]\!] = \mathit{sequnion}\ (\mathcal{F}[\![\mathrm{E}_0]\!])(\mathcal{F}^*[\![\mathrm{E}^*]\!])$$

$$\mathit{sequnion} : \mathrm{Ide}^* \to \mathrm{Ide}^* \to \mathrm{Ide}^*$$

$$\mathit{sequnion} = \lambda \mathrm{I}_1^*\mathrm{I}_2^*.\ \mathit{sequndif}\ \mathrm{I}_1^*\mathrm{I}_2^*\mathrm{I}_2^*$$

$$\mathit{sequnion} : \mathrm{Ide}^* \to \mathrm{Ide}^* \to \mathrm{Ide}^*$$

$$\mathit{seqdif} = \lambda \mathrm{I}_1^*\mathrm{I}_2^*.\ \mathit{sequndif}\ \mathrm{I}_1^*\mathrm{I}_2^*\langle\ \rangle$$

$$\mathit{sequnion} : \mathrm{Ide}^* \to \mathrm{Ide}^* \to \mathrm{Ide}^* \to \mathrm{Ide}^*$$

$$\begin{aligned}\mathit{sequndif} = \lambda \mathrm{I}_1^*\mathrm{I}_2^*\mathrm{I}_3^*.\ & (\#\mathrm{I}_1^* = 0) \to \mathrm{I}_3^*,\\ & (\mathit{inseq}\ \mathrm{I}_2^*\ (\mathrm{I}_1^* \downarrow 1)) \to \mathit{sequndif}\ (\mathrm{I}_1^*\dagger 1)\mathrm{I}_2^*\mathrm{I}_3^*,\\ & \langle \mathrm{I}_1^* \downarrow 1\rangle \ \S\ (\mathit{sequndif}\ (\mathrm{I}_1^*\dagger 1)\mathrm{I}_2^*\mathrm{I}_3^*)\end{aligned}$$

$$\mathit{inseq} : \mathrm{Ide}^* \to \mathrm{Ide} \to \mathrm{B}$$

$$\begin{aligned}\mathit{inseq} = \lambda \mathrm{I}^*\mathrm{I}.\ & (\#\mathrm{I}^* = 0) \to \mathit{false},\\ & ((\mathrm{I}^* \downarrow 1) = \mathrm{I}) \to \mathit{true},\\ & \mathit{inseq}\ (\mathrm{I}^*\dagger 1)\ \mathrm{I}\end{aligned}$$

### Primitive Procedures

All primitive procedures are in domain F.

$$\begin{aligned}\mathit{tproc?} = \mathit{onearg}\ (\lambda\nu.\ & (\nu \in \mathrm{T}) \to \mathit{return}\ (\mathit{true\ in}\ \mathrm{V}),\\ & \mathit{return}\ (\mathit{false\ in}\ \mathrm{V}))\end{aligned}$$

$$\begin{aligned}\mathit{tproc/decompose} = \mathit{onearg}\ (\lambda\nu.\ & (\nu \notin \mathrm{T}) \to \mathit{wrong},\\ & \mathit{return}\ (\nu|\mathrm{T}\ \mathit{in}\ \mathrm{V}))\end{aligned}$$

$$\begin{aligned}\mathit{tproc/make} = \mathit{twoarg}\ (\lambda\nu_1\nu_2.\ & (\nu_1 \notin \mathrm{Exp}) \to \mathit{wrong},\\ & (\nu_2 \notin \mathrm{REnv}) \to \mathit{wrong},\\ & \mathcal{E}_\lambda[\![\nu_1]\!](\mathit{reflectenv}\ (\nu_2|\mathrm{REnv})))\end{aligned}$$

$$\begin{aligned} selectexpr = onearg\ (\lambda\nu.\ (\nu \notin \text{Clo}) &\rightarrow wrong, \\ &return\ ((\nu|\text{Clo}) \downarrow 1)\ in\ \text{V}) \end{aligned}$$

$$\begin{aligned} selectenv = onearg\ (\lambda\nu.\ (\nu \notin \text{Clo}) &\rightarrow wrong, \\ &return\ ((\nu|\text{Clo}) \downarrow 2)\ in\ \text{V}) \end{aligned}$$

$$\begin{aligned} primitive? = onearg\ (\lambda\nu.\ (\nu \in \text{P}) &\rightarrow return\ (true\ in\ \text{V}), \\ &return\ (false\ in\ \text{V})) \end{aligned}$$

$$\begin{aligned} primitive/name = onearg\ (\lambda\nu.\ (\nu \notin \text{P}) &\rightarrow wrong, \\ &return\ (((\nu|\text{P}) \downarrow 1)\ in\ \text{V})) \end{aligned}$$

$$\begin{aligned} symeq = twoarg\ (\lambda\nu_1\nu_2.\ &(\nu_1 \notin \text{Ide}) \rightarrow wrong, \\ &(\nu_2 \notin \text{Ide}) \rightarrow wrong, \\ &(\nu_1 = \nu_2) \rightarrow return\ (true\ in\ \text{V}), \\ &return\ (false\ in\ \text{V})) \end{aligned}$$

**Primitive Utilities**

$onearg : (\text{V} \rightarrow \text{K} \rightarrow \text{V}) \rightarrow \text{F}$

$$\begin{aligned} onearg = \lambda\zeta\nu^*\kappa.\ &(\#\nu^* \neq 1) \rightarrow wrong, \\ &\zeta(\nu^* \downarrow 1)\kappa \end{aligned}$$

$twoarg : (\text{V} \rightarrow \text{V} \rightarrow \text{K} \rightarrow \text{V}) \rightarrow \text{F}$

$$\begin{aligned} twoarg = \lambda\zeta\nu^*\kappa.\ &(\#\nu^* \neq 2) \rightarrow wrong, \\ &\zeta(\nu^* \downarrow 1)(\nu^* \downarrow 2)\kappa \end{aligned}$$

$binary : (\text{N} \rightarrow \text{N} \rightarrow \text{N}) \rightarrow \text{F}$

$$\begin{aligned} binary = \lambda\zeta.\ twoarg\ (\lambda\nu_1\nu_2\kappa.\ &(\nu_1 \notin \text{N}) \rightarrow wrong, \\ &(\nu_2 \notin \text{N}) \rightarrow wrong, \\ &\kappa\ (\zeta\ (\nu_1|\text{N})\ (\nu_2|\text{N})\ in\ \text{V})) \end{aligned}$$

$binpred : (\text{N} \rightarrow \text{N} \rightarrow \text{B}) \rightarrow \text{F}$

$$\begin{aligned} binpred = \lambda\zeta.\ twoarg\ (\lambda\nu_1\nu_2\kappa.\ &(\nu_1 \notin \text{N}) \rightarrow wrong, \\ &(\nu_2 \notin \text{N}) \rightarrow wrong, \\ &\kappa\ (\zeta\ (\nu_1|\text{N})\ (\nu_2|\text{N})\ in\ \text{V})) \end{aligned}$$

$reflectenv : \mathrm{Bin}^* \to \mathrm{U}$

$$\begin{aligned} reflectenv = \lambda\beta^*.\ & (\#\beta^* = 0) \to (\lambda \mathrm{I}.\ Undefined), \\ & extend\,(reflectenv(\beta^*\dagger 1)) \\ & \qquad \langle((\beta^* \downarrow 1) \downarrow 1)\rangle \\ & \qquad \langle((\beta^* \downarrow 1) \downarrow 2)\rangle \end{aligned}$$

$\mathcal{E}_\lambda[\![\mathrm{K}]\!] = wrong$

$\mathcal{E}_\lambda[\![\mathrm{I}]\!] = wrong$

$\mathcal{E}_\lambda[\![(\mathrm{E}_0\ \ \mathrm{E}^*)]\!] = wrong$

$\mathcal{E}_\lambda[\![(\texttt{lambda}\ \ (\mathrm{I}^*)\ \ \mathrm{E}_0)]\!] = \lambda\rho.\ enclose\ \rho\ \mathrm{I}^*\ \mathrm{E}_0$

$\mathcal{E}_\lambda[\![(\texttt{if}\ \ \mathrm{E}_0\ \ \mathrm{E}_1\ \ \mathrm{E}_2)]\!] = wrong$

**Initial environment**

$$\begin{aligned} \rho_0 = \lambda \mathrm{I}.\ & (\mathrm{I} = \texttt{tproc?}) \to \langle \texttt{tproc?} | tproc?\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{tproc/make}) \to \langle \texttt{tproc/make} | tproc/make\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{tproc/decompose}) \to \langle \texttt{tproc/decompose} | tproc/decompose\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{->expr}) \to \langle \texttt{->expr} | selectexpr\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{->env}) \to \langle \texttt{->env} | selectenv\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{primitive?}) \to \langle \texttt{primitive?} | primitive?\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{primitive/name}) \to \langle \texttt{primitive/name} | primitive/name\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{Ide=?}) \to \langle \texttt{Ide=?} | symeq\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{call?}) \ldots, \\ & (\mathrm{I} = \texttt{make-call}) \ldots, \\ & (\mathrm{I} = \texttt{call/operator}) \ldots, \\ & (\mathrm{I} = \texttt{call/operands}) \ldots, \\ & (\mathrm{I} = \texttt{lambda?}) \ldots, \\ & (\mathrm{I} = \texttt{make-lambda}) \ldots, \\ & (\mathrm{I} = \texttt{lambda/bvl}) \ldots, \\ & (\mathrm{I} = \texttt{lambda/body}) \ldots, \\ & \ldots, \\ & (\mathrm{I} = \texttt{+}) \to \langle \texttt{+} | (binary\ +)\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{-}) \to \langle \texttt{-} | (binary\ -)\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{*}) \to \langle \texttt{*} | (binary\ *)\rangle\ in\ \mathrm{V}, \\ & (\mathrm{I} = \texttt{=}) \to \langle \texttt{=} | (binpred\ \ =_{\mathrm{N}})\rangle\ in\ \mathrm{V}, \\ & \ldots \end{aligned}$$

# Bibliography


[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley series in Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1974.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[4] Gail Anderson and Paul Anderson. *The Unix C Shell Field Guide.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] Emil Artin. *Galois Theory.* University of Notre Dame Press, 1942.

[6] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[7] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland Publishing Company, 1984.

[8] Alan Bawden. Reification without evaluation. In *Conf. on Lisp and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988. ACM.

[9] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[10] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[11] William Clinger and Jonathan (*editors*) Rees. Revised$^4$ report on the algorithmic language scheme. Technical Report AI Memo 848b, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1991.

[12] William Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX School/Workshop on the Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, pages 151–178, New York, NY, 1990. Springer-Verlag.

[13] Germumd Dahlquist and Björck Ake. *Numerical Methods.* Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[14] L. Peter Deutsch and Allan Schiffman. Efficient implementation of the smalltalk-80 system. In *Proc. 11th Principles of Programming Languages*, pages 297–302. ACM, 1983.

[15] G. Dowek. Third-order matching is decidable. Technical report, INRIA-Rocquencourt, 1991.

[16] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. Technical report, INRIA-Rocquencourt, 1991.

[17] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[18] Bruce W. Char et al. *Maple V Language Reference Manual.* Springer-Verlag, New York, NY, 1991.

[19] D. Weise et al. Automatic online partial evaluation. In *Conference on Functional Programming Languages and Architectures.* Springer Verlag, 1991.

[20] E. Charniak et al. *Artificial Intelligence Programming, 2nd ed.* Lawrence Erlbaum Assoc., Hillsdale, NJ, 1987.

[21] Marc Feeley and Guy Lapalme. Using closures for code generation. *Comput. Lang.*, 12(1):47–66, 1987.

[22] Matthias Felleisen and Hieb Robert. The revised report on the syntactic theories of sequential control and state. Technical Report TR-345, Indiana University Comp. Sci. Dept., February 1992.

[23] Amy Felty. A logic programming approach to implementing higher-order term rewriting. In *Second Intl. Workshop on Extensions of Logic Programming*, New York, NY, January 1991. Springer-Verlag.

[24] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Symp. on Lisp and Functional Programming*, pages 348–355, Austin, Tex., August 1984. ACM.

[25] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[26] Jean H. Gallier. Dpda's in "atomic normal form" and applications to equivalence problems. *Theoretical Computer Science*, (14):155–186, 1981.

[27] Leonard Gilman and Allen J. Rose. *APL, An Interactive Approach, 3rd ed.* John Wiley & Sons, New York, NY, 1984.

[28] W. D. Goldfarb. The undecidability of the second order unification problem. *Theoretical Computer Science*, (13):225–230, 1981.

[29] Michael J.C. Gordon. *The Denotational Description of Programming Languages, an Introduction.* Springer-Verlag, New York, NY, 1979.

[30] Mathlab Group. *Macsyma Reference Manual.* Mass. Inst. of Technology, Laboratory for Computer Science, Cambridge, MA, 1977.

[31] Chris Hanson. MIT Scheme reference manual. Technical Report TR-1281, Mass. Inst. of Technology, Artificial Intelligence Laboratory, November 1991.

[32] J. Heering. Implementing higher-order algebraic specifications. Technical Report CS-R9150, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, December 1991.

[33] Peter Henderson. Functional geometry. In *Symp. on Lisp and Functional Programming*, pages 1–9, Pittsburgh, PA, August 1982. ACM.

[34] James M. Henle and Eugene M. Kleinberg. *Infinitesimal Calculus.* MIT Press, Cambridge, Massachusetts, 1979.

[35] I. N. Herstein. *Topics in Algebra, 2nd ed.* John Wiley & Sons, New York, NY, 1975.

[36] G. Huet. *Résolution d'Équations dans les Langages d'Ordre 1,2,...,$\omega$.* PhD thesis, Université de Paris 7, 1976.

[37] Suresh Jagannathan. Reflective building blocks for modular systems. In *Intl. Workshop on Reflection and Meta-Level Architecture*, pages 61–68, November 1992.

[38] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21:93–124, March 1989.

[39] Donald E. Knuth. *The Art of Computer Programming, vol. 2: Seminumerical Algorithms, 2nd ed.* Addison-Wesley, Reading, Massachusetts, 1981.

[40] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, April 1974. Proc. Symposium on Very High Level Languages.

[41] R. G. Loeliger. *Threaded Interpretive Languages.* Byte Books, 1981.

[42] William A. Martin. Determining the equivalence of algebraic expressions by hash coding. *Journal of the ACM*, 18(4):549–558, October 1971.

[43] Dale Miller. Abstract syntax and logic programming. Technical Report MS-CIS-91-72, Univ. of Pennsylvania, Dept. of Computer and Information Science, October 1991.

[44] Dale Miller. Unification of simply typed lambda-terms as logic programming. Technical Report MS-CIS-91-24, Univ. of Pennsylvania, Dept. of Computer and Information Science, 1991.

[45] Dale A. Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. *Acta Informatica*, 11(1):31–55, 1978.

[46] Dale A. Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. Technical Report MS-CIS-85-17, Univ. of Pennsylvania, Dept. of Computer and Information Science, 1985.

[47] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[48] Gopalan Nadathur and Dale Miller. An overview of $\lambda$prolog. In *Fifth Intl. Conf. and Symposium on Logic Programming*, pages 810–827.

[49] William H Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge Univ. Press, Cambridge, UK, 1986.

[50] G. Rayna. *REDUCE—Software for Algebraic Computation*. Springer-Verlag, New York, NY, 1987.

[51] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975.

[52] Gerald Roylance. Expressing mathematical subroutines constructively. In *Conf. on Lisp and Functional Programming*, pages 8–13, Snowbird, Utah, July 1988. ACM.

[53] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Mass. Inst. of Technology, 1982. Available as TR-272 from the MIT Laboratory for Computer Science.

[54] Brian Cantwell Smith. Reflection and semantics in lisp. Technical Report CSLI-84-8, Stanford University Center for the Study of Language and Information, December 1984.

[55] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Technical Report Memo 380, Mass. Inst. of Technology, Artificial Intelligence Laboratory, September 1976.

[56] Richard Statman and Gilles Dowek. On statman's finite completeness theorem. Technical Report CMU-CS-92-152, Carnegie Mellon University, June 1992.

[57] Guy Lewis Steele Jr. *Common LISP The Language, 2nd Edition*. Digital Press, 1990.

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

[59] Gilbert Strang. *Linear Algebra and Its Applications, 3rd ed.* Harcourt Brace Jovanovich, San Diego, 1988.

[60] Bjarne Stroustrup. *The C++ Programming Language, 2nd ed.* Addison-Wesley, Reading, Massachusetts, 1991.

[61] Amin Vahdat. The design of a metaobject protocol controlling behavior of a scheme interpreter. Technical report, Xerox PARC, 1993.

[62] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.

[63] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, Reading, Massachusetts, 1988.