Lecture Notes on the Principles of Programming Languages

Shriram Krishnamurthi and Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77005-1892

October 14, 1997

Contents

Introduction

This document grew out of lectures given in Rice University's course COMP 311, Principles of Programming Languages, in the Spring semester of 1995-96. The course was taught by Matthias Felleisen.

The notes are freely available to everyone who wishes to study or teach the principles of programming languages.

Acknowledgments

We thank the students in the class as well as readers from outside Rice who have pointed out errors and suggested improvements. We especially thank Michael Ernst and Bruce Duba for pointing out mistakes and for useful discussions.

1 Studying Programming Languages

Guten morgen! Welcome to this exploration of the landscape of programming languages.

There are several key theses that we can formulate about the use and nature of programming languages. For instance:

Thesis 1: *Speak the programming language that you need to work with.*

Over the past few decades, thousands of programming languages have been designed, but programming language design is by no means a dead area. In only the past few years, a number of new languages have been developed and have become prominent, including:

- Perl
- HTML
- Java
- AMPL
- Active VRML

Each of these meets some specialized goal. For instance, AMPL is designed for expressing mathematics; HTML is a mark-up language for hypertext documents; and Active VRML, which was derived from the CAML family of languages, has been designed by Microsoft to enable transmission of active virtual reality scenarios across networks.

Thesis 2: *Programming languages are invented while you sleep, and spread before you wake up.*

This proliferation of languages has made it especially important to understand the design and functionality of languages. In particular, many goals already have languages designed to address them; the user only needs to find the appropriate language.

Thesis 3: *Understanding programming languages is the key to your job.*

This is a much more controversial statement, and hence bears elaboration. Why do we need to understand programming languages?

- Technical jobs in computer science will inevitably involve programming, which requires us to understand the languages we use.
- We might be called upon to choose a programming language for some project. Making such a selection involves several issues, including technological, sociological (such as training programmers) and economic (such as re-use of existing code and programming environments) considerations. Of these, we will focus mostly on technological considerations.
- Finally, we might be in a position to build a new language. To be successful, we must understand past efforts, current needs and key technological ideas.

At the heart of these issues is a fundamental question: *What does it mean to understand a programming language?* Let us illustrate the problem via an example. Consider the following statement:

set $x[i]$ *to* $x[i] + 1$

This is clearly intended to denote the increment of an array element. How would we translate this statement to a variety of different languages, and what would it mean?

In C (*circa* 1970) [6], we would write this as

$x[i] = x[i] + 1;$

This performs a hardware lookup for the address of x and adds i to it. The addition is a hardware operation, so it is dependent upon the hardware in question. This resulting address is then referenced (if it's legal — which it might not be), 1 is added to the bit-string stored there (again, as a hardware addition, which can overflow), and the result is stored back to that location. However, no attempt has been made to determine that x is even a vector and that $x[i]$ is a number.

In Scheme (1975) [1], this would be transcribed as

(*vector-set!* x i (+ (*vector-ref* x i) 1))

This does all the things the corresponding C operation does, but in addition it also (a) checks that the object named x is indeed an array, (b) makes sure i is within the bounds of the array, (c) ensures the dereferenced location contains a number, and (d) performs abstract arithmetic (so there will be no "overflow").

Finally, in Java (*circa* 1991) [5], one might write

$x[i] = x[i] + 1;$

which looks identical to the C code. However, the actions performed are those performed by the Scheme code, with one major difference: the arithmetic is not as abstract. It is defined to be done as if the machine were a 32-bit machine, which means we can always determine the result of an operation, no matter which machine we execute the program on, but we cannot have our numbers grow arbitrarily large.

Thus, we have a table as follows:

Operation	**Abstraction Level**		
	Hardware	**…**	**Abstract**
Vector lookup	C/C++		Java, Scheme, SML
Arithmetic	C/C++	Java, SML	Scheme

Note that the SML definition leaves the details of arithmetic unspecified, so some implementations provide abstract arithmetic while others offer machine-based numbers (or, possibly, both).

What do we need to know to program in a language? There are three crucial components to any language. The *syntax* of the language is a way of specifying what is legal in the phrase structure of the language; knowing the syntax is analogous to knowing how to spell and form sentences in a natural language like English. However, this doesn't tell us anything about what the sentences mean.

The second component is the meaning, or *semantics*, of a program in that language. Ultimately, without a semantics, a programming language is just a collection of meaningless phrases; hence, the semantics is the crucial part of a language.

Finally, as with natural languages, every programming language has certain *idioms* that a programmer needs to know to use the language effectively. This is sometimes referred to as the *pragmatics* of the language. Idioms are usually acquired through practice and experience, though research over the past few decades has lead to a better understanding of these issues.

Unfortunately, since the syntax is what a programmer first comes into contact with, and continues to overtly deal the most with, there is a tendency to over-emphasize the syntactic aspects of a language. Indeed, a speaker at a conference held in Houston in 1968 declared that the field of programming languages was dead, since we had understood everything about languages; the speaker was (largely) correct in referring to the syntactic problems that we must solve, but was failing entirely to consider the semantic issues involved.

There are several ways in which we can approach the study of languages. For instance, we could learn a little each of several languages that differ in some important aspect or another. There are several shortcomings in such an approach: it is hard to make direct comparisons, since by changing languages we may be changing several parameters; also, one would have to become comfortable with several different syntaxes and environments in very short order. To avoid these difficulties, we prefer to start with a single language that we define, which can then be enhanced in tightly-controlled ways as desired.

Having decided what to study, we must concern ourselves with how we will specify semantics. A natural language like English is a candidate for expressing semantics. However, natural languages are inherently imprecise; they are also unwieldy for expressing intricate details. (Witness, for instance, the descriptions of array incrementing above.) We can be precise in mathematical notation or, just as well, in an existing programming language; the latter offers the advantage of being an executable specification. Therefore, we choose to write programs which evaluate representations of programs in our defined language. Such programs are called *interpreters*. We prefer to write our interpreters in Scheme because the language makes it easy to prototype new languages that look syntactically similar to it.

Thus, we can characterize this course in contrast to synonymous offerings at most other institutions by the following matrix:

	Study by Breadth	**Study in Depth**
Natural Languages	Other courses	
Definitional Interpreters		This course

The next question of import is, *How does one choose a programming language?* More specifically, one might ask, why would one choose C for a task? There are some possibilities: it might offer some advantages in real-time systems, or it can often run with a small memory footprint. These criteria are especially valuable for systems that run in constrained environments and control critical machinery (be it a car or a missile).

Why would one use Scheme or Java? In addition to various language features that they offer, these languages also have the advantage of locating bugs sooner than the corresponding C programs. This is because each operation checks for errors, so an error is caught close to its physical location (though, of course, this may still be quite removed from the location of the logical error). Hence, it is impossible that the program will proceed blissfully with the wrong values, or terminate without having signaled an error at all. Hence, *ceteris paribus*, there is a clear likelihood of finding more errors, and sooner, in such languages. Detecting errors early is important for keeping the cost of development down. This shows how technology can have a direct impact on economics.

There is one final question that we must consider: *How do we evaluate programming languages?* We evaluate them by studying the details of their semantics to understand the universal properties and considering how each language treats these properties. For instance, two properties that are the subject of much current investigation are the type structure and memory management of programs. These properties give us coordinates along which languages may be classified.

2 Parsing

Syntax is the Viet Nam of programming languages.
—Matthias Felleisen

Last time, we discussed why it is important to understand programming languages, and how we propose to undertake this task. First, we need to understand the syntactic structure of programs.

2.1 Lexical Analysis

Consider this sequence of characters:

```
t hed oga tet hec at
```

It doesn't take long to recognize the above characters as spelling out the sentence, "The dog ate the cat". The process of taking an input stream of characters and converting it into a sequence of distinct, recognizable words is called *tokenizing* (or *lexical analysis*). Likewise, in Scheme, we might tokenize (**set!** x (+ x 1)) as (, **set!**, x, (, +, x, 1,), and).

The process of tokenizing is fairly straight-forward; determining the meaning of a sequence of tokens is much harder. To determine meaning, it helps to first determine what the legal sentences in the language are. We start by determining the phrases. Consider

$x = x + 1\ ;$

We have, in order, an identifier, the assignment symbol, another identifier, a binary operator, a literal and a terminator. We expect the assignment operator to be followed by an expression. An expression, in turn, can be two expressions separated by a binary operator, such as +. Finally, statements end with a terminator. Reasoning thus, we can conclude that the above sequence of characters represents a legal program phrase.

It's no accident that the semicolon looks like a claw, and terminates things.
—Larry Wall

2.2 Parsing: Is It a Legal Phrase?

In general, legal tokens don't always group into legal expressions, just as in English. The act of determining which sequence of tokens are put together legally, and of classifying these groups, is called *parsing*.

The potential inputs (*PI*) are all sequences that can be typed in. Of these, we want to determine the subset of syntactically legal phrases. (However, we aren't considering whether these phrases make *sense* yet.) We will use the following conventions: *num* represents numbers, (*PI PI*) represents a pair of *PI*'s, and (*PI* ...) is a list of *PI*'s. We will assume the set of potential inputs is

$$
\begin{aligned}
PI ::=\ & num \\
& \mid + \\
& \mid (PI \ldots)
\end{aligned}
$$

of which the legal inputs are

$$
\begin{aligned}
AE ::=\ & num \\
& \mid (AE + AE)
\end{aligned}
$$

For example, (1 2) is a *PI* but not an *AE*. Determining this is the job of a parser. Here's a program that performs this determination:

```
(define Parse
  (lambda (pi)
    (cond
      ((number? pi) #t)
      ((eq? '+ pi) #f)
      ((cons? pi) (if (and (= (length pi) 3)
                           (eq? (cadr pi) '+))
                      (and (Parse (car pi)) (Parse (caddr pi)))
                      #f)))))
```

2.3 Parsing: Abstract Syntax Trees

Consider the following syntaxes for assignment:

```
x = x + 1
x := x + 1
(set! x (+ x 1))
```

Which of these do we prefer?

The answer is that it really doesn't (and shouldn't) matter. All three perform the same abstract operation and differ only in surface syntax; thus, when we study the operations, we would like to do so without paying attention to surface details.

To abstract away the irrelevant syntactic details, we can create a data structure that represents the abstract operation. For instance, for the assignment code above, we might create

(*make-assignment-representation* <*representation of* "x">
<*representation of* "(x + 1)">)

The idea is to create *abstract syntax* that represents the important parts of the surface syntax. Abstract syntax is a language that

- represents the essence of groups of tokens, and
- makes it easy to distinguish between these groups.

For example, we might want to distinguish between unary and binary operations. Some operators such as − are both, so merely looking at the operator doesn't tell us what kind of operation we have; we would have to see how many operands it has been given. This is both inconvenient and inefficient. Instead, we can immediately distinguish between the uses of − by creating different abstract syntax representations:

(− 1) represented as (*make-unary-op* '− 1)
(− 1 2) represented as (*make-binary-op* '− 1 2)

Let us consider a concrete example of parsing a surface syntax into abstract syntax. Assume the following potential inputs (*PI*) and legal programs (*L*):

PI ::= *Num* | *Sym*
| (*PI* ...)

$L ::= Num \mid Var$
$\quad \mid (L\ L)$
$\quad \mid (\text{'lambda}\ Var\ L)$

where *Var* is the set *Sym* sans 'lambda. Names that are specified explicitly (like **lambda**) are called *keywords*. They are markers that distinguish a group of tokens.

We will use Scheme's data-definition language to design abstract representations of legal programs. Here is the set of data constructors that we will use:

```
(define-structure (num n))
(define-structure (var s))
(define-structure (proc param body))
(define-structure (app rator rand))
```

and our parser can construct appropriate records after examining each record for its legality.

```
(define Parse
  (lambda (pi)
    (cond
      ((number? pi) (make-num pi))
      ((symbol? pi) (if (var? pi) (make-var pi) (error 'parse)))
      (else
        (cond
          ((= (length pi) 2)
           (make-app (Parse (car pi)) (Parse (cadr pi))))
          ((and (= (length pi) 3)
                (eq? 'lambda (car pi))
                (and (symbol? (cadr pi)) (var? (cadr pi))))
           (make-proc (cadr pi) (Parse (caddr pi))))
          (else (error 'parse)))))))
```

Be sure to understand why we call *error* rather than return #f when we encounter illegal syntax.

Exercise: How can we construct a parser that signals more than one error?

We can now summarize our view of syntax:

- The stream of input characters is tokenized.
- The parser converts tokens into abstract representations. In the process, it
  - checks the legality of the input, and
  - hides irrelevant syntactic clutter.

If you design or build programming languages, you will have to understand much more about constructing and parsing surface syntax, and on the psychological impact of your choices. Our approach has some failings:

- We didn't deal with the process of tokenization.

- Grouping isn't a very difficult problem, though it can often be more complex than our presentation might suggest.
- Most language syntaxes are far more complex than the ones we saw; hence, the process of determining whether a given input is syntactically legal can be much harder.
- Most of all, the process of detecting and signaling errors is fairly difficult.

These problems are considered in detail in compiler construction courses. For our purposes, this treatment is sufficient.

3 Free and Bound Variables, Scope and Static Distance

Almost every programming language has constructs for introducing a new variable and for describing which occurrences of the variable in the rest of the program refer to this particular variable. Thus if a variable is used several times, a programmer can resolve which variable is really meant and can determine the current value of the variable.

For example, in C++ a loop header may introduce a new loop variable and its scope:

$$
\begin{aligned}
&\mathit{for}(\mathit{int}\ i = 0; \\
&\qquad i < \mathit{NumOfElements}; \\
&\qquad i{+}{+}) \ldots ;
\end{aligned}
$$

The scope of the variable i is the program text following the $=$ all the way to the end of the loop body. If the loop contains a break construct and if the loop is followed by some statement like

printf("the index where we stopped searching is %d\n", *i*);

then we know from the rules of scoping that the program probably contains an error. Since the i in the call to *printf* does *not* refer to the i that was introduced as the loop variable but to some i that was declared elsewhere, it is unlikely that the value of i in the call to *printf* is that of the loop index when the loop was terminated.

Note that we reasoned about the program with a vague understanding of its exact execution but with a precise understanding of the scoping rules. These rules are crucial for building a good mental execution model.

Other examples of scoping constructs in C++ include

- function definitions
- parameter declarations
- *::* for adding methods to a class
- blocks, which are introduced by declarations and via braces

---

Exercise: Determine in a C++ book what scope these constructs establish.

3.1 Free and Bound, Scope: A Simple Example

Using LC, our little toy language from the previous section, we can illustrate how to specify the notion of free and bound occurrences of program variables rigorously. This specification will serve two purposes:

- it will make the concepts *free*, *bound*, and *scope* more concrete for a small, simple language;
- it will show how to use English to define concepts rigorously.

Recall the surface syntax of LC:

$$
\begin{aligned}
&\mathit{Exp} ::= \mathit{Var} \mid \mathit{Num} \mid (\textbf{lambda}\ \mathit{Var}\ \mathit{Exp}) \mid (\mathit{Exp}\ \mathit{Exp}) \\
&\mathit{Var} ::= \mathit{Sym} \setminus \{'\mathit{lambda}\}
\end{aligned}
$$

If we interpret LC as a sub-language of Scheme, it contains only one binding construct: **lambda**-expressions. In

(**lambda** *a-var an-exp*)

a-var is introduced as a new, unique variable whose scope is roughly the entire **lambda**-expression, which effectively is *an-exp*.

One way to determine the scope of variable binding constructs is to define when some variable occurs free in some expression. By specifying where a variable occurs free and how this relation is affected by the various constructs, we also define the scope of variable definitions. Here is the definition for LC.

Definition 1 (Free occurrence of a variable in LC) *Let a-var, another-var range over the elements of Var. Let a-num range over the elements of Num. Let an-exp, another-exp range over the elements of Exp.*

Then,

- *The variable a-var occurs free in the expression another-var if a-var = another-var.*
- *The variable a-var does not occur free in the expression a-num.*
- *The variable a-var occurs free in the expression* (**lambda** *another-var an-exp*) *if a-var ≠ another-var* and *if a-var occurs free in an-exp.*
- *The variable a-var occurs free in the expression* (*an-exp another-exp*) *if it occurs free either in an-exp or in another-free.*
- *The variable a-var does not occur free elsewhere.*

A variable *a-var* occurs bound in the expression *an-exp* if it occurs as the second component of a **lambda**-expression. This particular occurrence of a variable is also known as the *binding occurrence*.

Note that the definition of the relation is inductive over expressions precisely because the definition of *Exp* is itself inductive.

Exercise: Formulate the notion of bound/binding occurrences explicitly as an inductively defined relation.

The definitions of free and bound implicitly determine the notion of a scope in LC expressions. Clearly, a **lambda** expression opens a new scope; or, to put it differently, the scope of the binding occurrence *a-var* in

(**lambda** *a-var an-Exp*)

is that textual region of *an-Exp* where *a-var* might occur free.

Exercise: Find an expression in which *x* occurs both free and bound.

3.2 Static Distance Representation

Given a variable occurrence it is natural for a programmer to ask where the binding occurrence is. This may tell him whether or not some nearby variable occurrence is related, or it may help him to determine something about the value that it stands for. Consider the expression

(**lambda** *z* (**lambda** *x* ((**lambda** *x* (*z* (*z* (*z x*)))) *x*)))

The two occurrences of *x* in the fragment

... *x*)))) *x*)

are unrelated, but only the binding occurrences for the two can tell them apart.

For a human being, a representation that includes arrows from bound occurrences to binding occurrences is clearly preferable. We can approximate such graphical representations of programs by replacing variable occurrences with numbers that indicate how far away in the surrounding context the binding construct is. The above expression would translate into

(**lambda** *z* (**lambda** *x* ((**lambda** *x* (3 (3 (3 1)))) 1)))

Indeed, since the parameters in **lambda**'s are now superfluous, we can omit them completely:

(**lambda** (**lambda** ((**lambda** (3 (3 (3 1)))) 1)))

This representation is often called the static distance representation of the term. Although this approximation of a graphical representation is not particularly helpful for people, it is valuable for compilers and interpreters.

We could specify the process that replaces variable occurrences with static distances in English along the lines of the above definitions. Instead, we write a program that performs the translation.

First, recall the abstract representation of the set of LC expressions

ARE ::= (*make-var Var*)
| (*make-const Num*)
| (*make-proc Var ARE*)
| (*make-app ARE ARE*)

based on the data definitions

(**define-structure** (*var name*))
(**define-structure** (*const num*))
(**define-structure** (*proc param body*))
(**define-structure** (*app rator rand*))

Second, the program template for abstract representations is clearly

```
(define fARE
  (lambda (an-are)
    (cond
      ((var? an-are) ...)
      ((a-const an-are) ...)
      ((proc? an-are) ... (fARE ... (proc-body an-are))
                      ... (proc-param an-are) ...)
      ((app? an-are) ... (fARE (app-rator an-are)) ...
                     ... (fARE (app-rand an-are)) ...))))
```

Since the replacement process substitutes a variable by a number that depends on the context of the variable, that is, the syntactic constructions surrounding the variable occurrence, we also need an accumulator. In our case, it suffices to accumulate the variables in binding constructs as we traverse the expression. This means the template can be refined to:

```
(define fARE
  (lambda (an-are binding-vars)
    (cond
      ((var? an-are) ...)
      ((a-const an-are) ...)
      ((proc? an-are)
       ... (fARE (proc-body an-are)
                 (cons (proc-param an-are) binding-vars)) ...)
      ((app? an-are) ... (fARE (app-rator an-are) binding-vars) ...
                     ... (fARE (app-rand an-are) binding-vars) ...))))
```

Finally, to distinguish the initial abstract representation of programs from the new one, we introduce two new data definitions:

```
(define-structure (sdvar sdc))
(define-structure (sdproc body))
```

We do not introduce new records for constants, because these don't change, and those for applications are structurally identical.

We can now complete the translation:

```
(define SD
  (lambda (an-are binding-vars)
    (cond
      ((var? an-are) (sdlookup (var-name an-are) binding-vars))
      ((const? an-are) an-are)
      ((proc? an-are)
       (make-sdproc
        (SD (proc-body an-are) (cons (proc-param an-are) binding-vars))))
      (else
       (make-app (SD (app-rator an-are) binding-vars)
                 (SD (app-rand an-are) binding-vars))))))
```

where

```
(define sdlookup
  (lambda (a-var lovars)
    (cond
      ((null? lovars) (error 'sdlookup "free occurrence of ~s" a-var))
      (else (if (eq? (car lovars) a-var)
                1
                (add1 (sdlookup a-var (cdr lovars))))))))
```

Variables are replaced by their static distance coordinate, which is determined by looking up how deep in the list of binding vars it occurs. If the list does not contain the variable, we signal an error.

Constants do not need to be translated. Applications are traversed and re-constructed.

We can test SD by applying it to the result of parsing some surface syntax and the empty list.

Exercise: What should the output of this be?

(*SD* (*Parse* '(lambda z (lambda x ((lambda x (z (z (z x)))) x)))) *null*)

(The empty list of variables indicates that we consider this to be the complete program with no further context.)

---

4 Syntactic Interpreter for LC (Behavior)

Recall the syntax of LC. We will add numerals and a binary operator, +, to the language, giving the following syntax for terms in the language:

$M ::= x \mid (\text{'}\mathsf{lambda}\ (x)\ M) \mid (M\ M)$
$\quad \mid n \mid (\text{'}+ M\ M)$

Think of LC as being a sub-language of Scheme.

Recall that in Comp 210, we formulated the semantics of Scheme as an extension to the familiar algebraic calculation process. Consider (+ (+ 3 4) (+ 7 5)). To evaluate this, we first determine the value of (+ 3 4) and of (+ 7 5). Why? Because these are not values and + can only add two values (numbers).

What is a *value*? We must first understand this concept to be able to explain the process of evaluation.

First, **lambda** terms are values. Are applications values? No, they are not; they trigger an evaluation. Is a number a value? Yes, it is. Finally, is a + term a value? No. In summary, $(M\ M)$ and $(+\ M\ M)$ are called *computations*, (**lambda** …) and numbers are *values*, and identifiers ... we leave these alone for now. Anyway, by following the regular rules of evaluation, we determine that the expression above yields the value 19. Let us consider each class of terms individually, but let us ignore variables for the moment.

4.1 Rules of Evaluation

Rule 1 *For +-expressions, evaluate the operands, then add them.*

Rule 2 *For applications, substitute the argument for the parameter in the body.*

Does this argument have to be a value? We shall require that it must. What happens if we don't require the argument to be a value? Certain computations which we might expect to not terminate do, in fact, yield a value. Consider

((**lambda** (y) 5) ((**lambda** (x) ($x\ x$)) (**lambda** (x) ($x\ x$))))

The argument given to the procedure does not reduce to a value. However, since the argument need not be evaluated before being substituted for y, the application can be reduced, evaluating to the answer 5.

Hence, we "patch" our application rule: *For applications, substitute the value of the argument for the parameter in the body.* However, the rule still isn't quite satisfactory. Consider the procedure (**lambda** (x) (**lambda** (x) x)). When this procedure is applied, we do not want the inner x to be substituted; that should happen only when the inner procedure is applied. Hence, we amend our rule to read, *For applications, substitute the value of the argument for all free instances of the parameter in the body.*

We are still not done. Consider the computation

((**lambda** (x) (+ $x\ x$)) 5)

The purpose of an evaluation rule is to yield a *value* in the end (if one exists); however, as per our current application rule, the result of reducing the computation above is (+ 5 5), which is not a value. Hence, we make the Third Amendment: *For applications, substitute the value of the argument for all free instances of the parameter in the body, and evaluate the resulting expression.*

There is yet more. Consider the following reduction:

  ((**lambda** (y) (**lambda** (x) y)) (**lambda** (z) x))
$=$ (**lambda** (x) (**lambda** (z) x))

Is there something strange about this? Yes! The outer x (in the argument) got bound by the inner (**lambda** (x) ...). This is clearly not at all what was intended. However, this is entirely consistent with our application rule.

Exercise: Write down a program that is very much like the one above that uses the evaluation rules given so far and produces the "wrong" result, but which uses the new rule (to be given) and yields the "right" result.

There are two ways out of this conundrum:

- We can institute a rule that programs cannot contain free variables. (In practice, only a few languages, such as Scheme, Lisp and SNOBOL, do not enforce such a restriction.)
- We can once again redefine our notion of substitution: our final amendment will require that we rename *all* binding instances (and their corresponding bound instances) before we perform the substitution. The new name chosen should be one that cannot yield such inadvertent capture (such as, say, a name not found anywhere else in the program). This technique is called "clean substitution" or "hygienic substitution" [7, 8].

If we wanted our language LC to resemble Scheme, we would have to use the second alternative above. At this point we deviate from Scheme and assume that a program does not contain any free variables.

4.2 An Evaluator

Now we can translate our rules into a program. Here is a sketch of the evaluator:

```
(define Eval
  (lambda (M)
    (cond
      ((var? M) (impossible ...))
      ((num? M) M)
      ((proc? M) M)
      ((add? M) (add-num
                   (Eval (add-left M))
                   (Eval (add-right M))))
      (else ; (app? M) ⟹ #t
        (Apply
          (Eval (app-rator M))
          (Eval (app-rand M)))))))
```

```
(define Apply
  (lambda (a-proc a-value)
    (Eval (substitute a-val ; for
              (proc-param a-proc) ; in
              (proc-body a-proc)))))
```

```
(define substitute
  (lambda (v x M)
    (cond ; M
      ... cases go here ...
      )))
```

The key property of this evaluator is that it only manipulates (abstract) syntax. It specifies the meaning of LC by describing how phrases in LC relate to each other. Put differently, the evaluator only specifies the behavior of LC phrases, roughly along the lines of a machine, though the individual machine states are programs. Hence, this evaluator is a *syntactic* method for describing meaning.

4.3 Toward Assigning Meaning to Phrases

This evaluator (*Eval*) has some problems. To understand these, consider its behavior on an input like

((**lambda** (*x*) <*big-proc*>) 5)

Assume that <*big-proc*> consists of many uses of *x*. The evaluator must step through this entire procedure body, substituting every occurrence of *x* with the value 5. The tree produced by this process is of the same size as the original tree (since we traversed the entire tree, and replaced an identifier with a value). What does *Eval* do next? It walks once again over the entirety of the same tree that it just produced after substitution, for the purpose of evaluating it. This is clearly very wasteful.

To be more frugal, *Eval* could instead do the following: it could merge the two traversals into one, which is carried out when evaluating the (as yet unsubstituted) tree. During this traversal, it could carry along a table of the necessary substitutions, which it could then perform just before evaluating an identifier. This table of delayed substitutions is called an *environment*.

Hence, our evaluator now takes two arguments: the expression and an environment. The environment *env* contains a list of the identifiers that are free in the body, and the values associated with them.

```
(define Eval
  (lambda (M env)
    (cond
      ((var? M) lookup M's name in env)
      ((num? M) ...)
      ((proc? M) what do we do here?)
      ((add? M) ...)
      (else))))
```

When we see a procedure, we need to keep track of the environment active at the time the procedure was evaluated. (This is because the procedure might not be applied immediately.) So we make the value of a procedure in LC be a combination (in the meta-language) of the body and the environment. This is called a *closure*. Doing this leads to a very important idea: that we should use entities in the meta-language that we understand well to represent the meaning of phrases in the language under consideration. In addition to representing procedures as closures, it is also natural to make LC numerals evaluate to Scheme numbers.

4.4 Summary

We have

- given most of the rudiments of a syntactic evaluation theory, and
- started to de meta-interpretation (sometimes known as studying "semantics", the process of using a meta-language with a well-understood meaning to specify the meaning of a language).

In these examples, we have been using Scheme as our meta-language.

5 Meta-Interpreter for LC (Meaning)

In summary, the previous section was about syntactic interpreters, which rewrite programs in the syntax of the source language. This is a powerful interpretation technique. For instance, even utilities as seemingly far removed from programming languages as the *sendmail* daemon use it for configuration files. In this section, we will look at meta-interpreters, which are used to denote meanings of phrases in a program.

A *meta-interpreter* represents procedures in LC as combinations (closures) of syntactic procedures and environments. The initial motivation for the name "meta" is that, instead of taking the program text and reducing it to new program text, we choose an element of the implementing (or 'meta') language to represent a phrase in the implemented language. A secondary association is that we interpret every construct as directly as possible in the interpreted language.

Last time, we had already decided to use combinations of procedure expressions and environments to interpret LC lambda expressions (closures). This also suggests that we use Scheme numbers to interpret LC numbers. The latter choice implies that we can interpret LC addition as Scheme addition. This in turn suggests the use of Scheme procedures for the interpretation of LC lambda expressions so that we can use application to interpret LC application.

Here is a sketch of *MEval*, which is *Eval* transformed according to our informal ideas about delaying substitutions and choice of representations:

```
(define MEval
  (lambda (M env)
    (cond
      ((var? M) (lookup (var-name M) env failure-contn))
      ((num? M) (num-num M))
      ((add? M) (+ (MEval (add-left M) env)
                   (MEval (add-right M) env)))
      ((proc? M) (make-closure M env))
      ((app? M) (MApply (MEval (app-rator M) env)
                   (MEval (app-rand M) env))))))
```

NOTE: The + operation used above must be chosen with care, since the addition operation in the meta-language won't necessarily be the same as that of the implemented language. Note also that we pass a *failure continuation* to the *lookup* procedure. The failure continuation is a procedure of no arguments that is invoked if the identifier cannot be found in the environment.

What are the values in LC? There are two: numerals and procedures. Numerals can be represented directly in the meta-language. To avoid a premature choice of representation for closures, we have chosen to use the abstractions *make-closure* and *MApply*. Thus, if we ever need to change the interpretation of closures, we can do so without changing the interpreter itself.

Exercise: Which elements of the interpreter would we have to change if we change one of our representation choices?

In the special case when the language we are interpreting is the same as that in which the interpreter is written (for instance, a Scheme interpreter written in Scheme), we call the interpreter *meta-circular*.

Let us examine the representation of procedures.

```
(define make-closure
  (lambda (proc-exp env)
    (lambda (value-for-param)
      (MEval (proc-body proc-exp)
        (extend env (proc-param proc-exp) value-for-param)))))
```

```
(define MApply
  (lambda (val-of-fp val-of-arg-p)
    (val-of-fp val-of-arg-p)))
```

Note that the closure returned by *make-closure* closes over *env*.

Abstractly, we can characterize *MApply* and *MEval* as follows:

```
  (MApply (make-closure (make-proc x B) Env) Val)
= (MEval B (extend Env x Val))
```

Exercise: Given the declaration

(**define-structure** (*closure P E*))

how do we write *MApply*?

5.1 Environment Representation

One part of the interpreter has still been left unspecified: the representation of environments. Before considering the available alternatives, it is worth-while to consider environments abstractly too. There are three things we need to understand with respect to environments: the lookup method, the extension method, and the empty environment. The latter two create new environments, while the first of these extracts information from information. Here are the equations that relate the constructors and the selector:

```
(lookup Var (mt-env) F) = (F)
(lookup Var (extend Env VarN Val) F) =
  (if Var is VarN
      Val
      (lookup Var Env F))
```

What is a good representation choice for environments? Note that there is only a fixed number of free variables in a given program, and that we can ascertain how many there are before we begin evaluating the program. On the other hand, we can be lax and assume that there can be arbitrarily many free variables. A good representation in the former case is the vector; in the latter case, we might wish to use lists. However, there is at least one more representation.

5.2 Representing Environments as Procedures

Consider the following implementations:

```
(define lookup
  (lambda (Var Env Fk)
    (Env Var Fk)))

(define mt-env
  (lambda ()
    (lambda (Var Fk)
      (Fk))))
```

We can then prove that this implementation satisfies one of the equations that characterize environments:

```
  (lookup var (mt-env) f)
= (lookup var ((lambda () (lambda (Var Fk) (Fk)))) f)
= (lookup var (lambda (Var Fk) (Fk)) f)
= ((lambda (Var Fk) (Fk)) var f)
= (f)
```

as desired. We can similarly define *extend*:

```
(define extend
  (lambda (Env VarN Val)
    (lambda (name Fk)
      (if (eq? name VarN)
        Val
        (Env name Fk)))))
```

Exercise: Verify that *extend* and *lookup* satisfy the above equation.

5.3 Binding Constructs

Now suppose we added some new binding constructs to LC. For instance, suppose we added *seq-let*, and defined its behavior as follows:

```
    (MEval "(seq-let Var RHS Body)" env)
==> (MEval Body (extend env Var
                        (MEval RHS env)))
```

However, now say we add recursive lexical bindings:

```
    (MEval "(rec-let Var RHS Body)" env)
==> (MEval Body (extend env Var
                        (MEval RHS ...)))
```

where the … represents the (*extend env Var* …) term. How can we implement such a construct? We clearly need a way to create an environment that refers to itself. If we represent environments as procedures, we can use recursive procedures to implement this kind of extension.

Exercise: Can we use lists or other representations to accomplish this goal?
Hint: What did we do in Comp 210 to create data structures that refer to themselves?

---

6 Recursive Definitions and Environments

Slogan: *Definitions build environments.*

Let us add a recursive binding mechanism to our language. We can represent its abstract syntax with the following data definition:

```
(define-structure (rec-let
                    lhs   ; variable
                    rhs   ; required to be a lambda-expression
                    body))
```

where the *lhs* is bound in both *rhs* and *body*. The code for it in the interpreter might look like

```
((rec-let? M) ... (Interp (rec-let-body M)
                    (extend env
                      (rec-let-lhs M)
                      (make-closure (rec-let-rhs M) E))))
```

To turn the *rhs* expression of *M* into a recursive closure, we desire that *E* be exactly like the environment that we are in the process of constructing. In other words, we would like a special kind of environment, *env*∗, with the following property:

env∗ = (*extend env* (*rec-let-lhs M*) (*make-closure* (*rec-let-rhs M*) *env*∗))

Using the following procedure

```
(define F
  (lambda (env*)
    (extend env (rec-let-lhs M) (make-closure (rec-let-rhs M) env*))))
```

the equation for *env*∗ can be rewritten as

env∗ = (*F env*∗)

which shows that we want the environment to be the *fixed-point* of the function *F* (from environments to environments).

Sidebar: Fixed-Points Over the Reals

Say we have functions from $\mathcal{R}$ to $\mathcal{R}$. We might have a function like

$$f(x) = 2x + 1$$

The fixed-point of f is the value x_f such that $x_f = f(x_f)$. Thus, we want $x_f = 2x_f + 1$, solving which we get the fixed-point $x_f = -1$.

Does every function have a fixed-point? No: consider $g(x) = x + 1$. Substituting x_f and reducing, we get $0 = 1$, which is a contradiction. On the other hand, $h(x) = x$ has an infinite number of solutions. Hence, a function from $\mathcal{R}$ to $\mathcal{R}$ could have zero, one or an infinite number of fixed-points. However, for every function from environments to environments, we can construct a fixed-point.

6.1 Environments via Functions

If environments are implemented as functions, the definition of *fix-env* is straightforward:

```
(define fix-env
  (lambda (F)
    (letrec ((renv
              (F (lambda (id fail-k)
                   (lookup id renv fail-k)))))
      renv)))
```

So we have used **letrec** to implement *rec-let*. Note that we have placed a strong requirement on the *rhs* of *rec-let* in LC. Therefore, this interpreter does not adequately explain how **letrec** works in Scheme.

Exercise: Do we need Scheme's **letrec** to interpret *rec-let*?
Hint: Take a look at Matthias Felleisen's *Lecture on the Why of Y* [2].

6.2 Environments via Structures

If we represent environments as structures, how do we write *fix-env*? It clearly means there is a circularity. What do we use to build circular data structures? We use **set!**. Hence, we could write *fix-env* in this manner instead:

```
(define fix-env
  (lambda (F)
    (let* ((senv (mt-env))
           (renv (F senv)))
      ; Is renv the appropriate environment yet? No.
      (set-alpha-env! senv (alpha-env renv))
      renv))) ; ... or senv; they are now equal.
```

(Note that in reality for this to work, we need to have some way of getting at the object which refers to the environment. In other words, we could put every environment inside a box. This gives us a handle on the environment, which can be referred to. Of course, now every environment operation would have to be sure to perform the unboxing operation to extract the actual environment.)

If we have multiple names, what can we do? We replace *extend* with *extend∗*. It can be written in terms of *extend*. But *fix-env* does not change at all.

6.3 Memory Management, Part 0

So far we have understood several things, such as procedures and applications, in terms of their Scheme counterparts. A notable exception is environments, which we have represented explicitly. Indeed, due to this explicit management of variable-value associations, we can analyze such lower-level aspects of the language as its memory management.

Consider the evaluation of

```
(let (x 1)
```

(**let** (y 2)
  (**let** (z 3)
    ...)))

We begin with the empty environment; as we encounter each **let**, we add a new level to our environment; as we leave the **let** body, we remove that level. At the end of the entire expression, our environment will be empty again. Hence, our environment will have behaved like a *stack*. Do environments always behave in this manner?

Consider this expression:

((**lambda** (x) (**lambda** (y) (y x))) 10)

This evaluates to the "apply to 10" function. This expression

((**lambda** (z) (**lambda** (w) ($+$ z w))) 20)

evaluates to a procedure that adds 20 to its argument. Apply the former to the latter. What happens to the environment at each stage?

We begin with the empty environment; the first application is performed and the environment is extended with the binding [x 10]. The closure is now created, and the environment is emptied. Then we evaluate the second expression, which adds the binding [z 20]. The second closure (call it *C2*) is produced, and the environment is emptied again.

At this point, we are ready to perform the desired application. When we do, the environment has the bindings [x 10] and [y *C2*]. Now *C2* is applied, which has the effect of replacing the current environment with its own, which contains the bindings [z 20]. The application adds the binding [w 10], at which point the addition is performed, 30 is returned, and the environment is emptied again.

The moral of this is that environments for LC programs, no matter how we choose to represent them, branch out like trees, and a simple stack discipline is insufficient for maintaining them. It is often the programmer's job to keep track of memory; in our presentation, we have left this task to Scheme's run-time system. Hence, this is another manner in which we have modeled by appealing to the underlying implementation in Scheme.

Exercise: Impose restrictions on LC so that it remains a procedural language yet does not require a tree-based management of environments.

---

7 Assignment and Mutability

We will add an assignment operation to the language, with the syntax

(**set!** *x M*)

and the abstract representation

(**define-structure** (*setter lhs rhs*))

where *x* can be any **lambda**-bound identifier. **set!** allows us to model changing events in the real world. Typically, a single **set!** enables us to create cycles and graphs, while multiple **set!**s model state in the modeled universe.

Consider the following code fragment:

```
((lambda (x)
 ...
 (set! x 6)
 ...)
 5)
```

What does the **set!** do to *x*? Initially, *x* is bound to the *value* 5. But the 5 is a constant, so it can't directly be changed (since the 5 can't be changed into a 6). Rather, we will need to change the value associated with *x*. We can accomplish this by altering the environment:

((*setter? M*) *change the environment*)

But how do we do this?

To make variables assignable, we need to change what they stand for. They cannot be directly associated with values; rather, they must be associated with *something*, which is associated with the value, and which can be changed to associate with a different value. What can these *something*s be? Boxes are a reasonable object to use here.

Moral: *Variables must stand for boxes.*

Where do we associate boxes with values? Since every use of a procedure is supposed to create a new copy of the body with all occurrences of the parameter replaced by the argument value, it is natural to create a box right after the argument is evaluated and to put the value of the argument into that box. Put technically, we change the interpretation of application as follows:

```
((app? M)
  (MApply
    (MEval (app-fp M) env)
    (box (MEval (app-ap M) env))))
```

This in turn induces another change in the interpreter: since every variable is boxed, we now have to unbox it to get its actual value.

```
((var? M)
  (unbox (lookup (var-name M) env fail-k)))
```

At this point, there is no LC program we can write to distinguish this implementation of LC from the one we had before.

We are now ready to interpret a **set!** expression:

```
((setter? M)
  (set-box! (lookup (setter-lhs M) env fail-k)
    (MEval (setter-rhs M) env)))
```

Why is the value returned by *MEval* not itself boxed? A **set!** expression only changes the value that a variable is associated with; it does not introduce a new variable.

7.1 Call-by-Value and Call-by-Reference

Consider this program, which contains a mutation:

```
(let ((f (lambda (x) (set! x 5))))
  (let ((y 10))
    (let ((◇ (f y)))
      y)))
```

What is the result of this? The value of *y*, 10, is placed in a *new box* when *f* is applied; this new box is thrown away after the procedure body (including the **set!**) has been evaluated, so the eventual value is that of *y*, which is still 10. This is *call-by-value*: we passed the *value* of *y*, not the capability to change its value. Therefore, in this language, we cannot write a *swap* procedure.

Assume we want that the following expression

```
(let ((f (lambda (x y)
           (let ((t x))
             (set! x y)
             (set! y t)))))
  (let ((a 5) (b 6))
    (f a b)
    (cons a b)))
```

evaluate to (*cons* 6 5). Then we have to pass *references* to variables *a* and *b*, not their *value* alone. We can accomplish this with a small change in the interpreter, which is motivated by a simple implementation-oriented observation: when the argument expression in an application is already a variable, it is associated with a box in the environment. Hence, we can pass this box to the procedure and don't need to create a new one:

```
((app? M)
  (MApply (... fp ...)
    (if (var? (app-ap M))
      (lookup (var-name (app-ap M)) env fail-k)
      (...))))
```

This new mode of parameter-passing is called *call-by-reference*. Pascal and Fortran IV used variants of this parameter-passing technique. While passing references enables programmers to write procedures like *swap*, it also introduces a new phenomenon into the language: variable aliasing.

Variable aliasing occurs when two syntactically *distinct* variables refer to the *same* mutable location in the environment. In Scheme such a coincidence is impossible; in Pascal it is common.

NOTE: A different form of aliasing, *data aliasing*, occurs when two distinct paths into a compound data structure refer to the same location. Both call-by-value and call-by-reference languages permit data aliasing.

7.2 Mutation: An Alternate Design

SML offers one middle ground between pass-by-value and pass-by-reference. It forces a programmer to associate variables that are used for modeling cycles or state change with reference cells (or boxes). Put differently, it makes references into values and can thus turn pass-by-value into the parameter passing of references exactly when needed.

To model SML we introduce three new classes of expressions:

- (*ref M*), which creates a ref cell, which is a record with one slot that holds the value of *M*;
- (*! M*), which assumes that *M* will evaluate to a ref cell; and,
- (*:= M1 M2*), which assumes *M1* will evaluate to a ref cell, and replaces the contents of that ref cell with the value *M2* reduces to.

In LC-SML, ref cells are a new class of values, distinct from numbers and procedures (closures). Interpreting the new expressions requires the addition of three new lines to the original interpreter of LC:

```
(define MEval
  (lambda (M env)
    (cond
      ((var? M) (lookup (var-name M) env failure-contn))
      ((num? M) (num-num M))
      ((add? M) (+ (MEval (add-left M) env)
                   (MEval (add-right M) right)))
      ((proc? M) (make-closure M env))
      ((app? M) (MApply (MEval (app-rator M) env)
                   (MEval (app-rand M) env)))
      ((ref? M)
        (box (MEval (ref-init M) env)))
      ((!? M)
        (unbox (MEval (!-expr M) env)))
      ((:=? M)
        (set-box! (MEval (:=-lhs M) env)
          (MEval (:=-rhs M) env))))))

(define MApply
  (lambda (val-of-fp val-of-arg-p)
    (val-of-fp val-of-arg-p)))
```

Exercise: Write the function *swap* in LC-SML.

7.3 Orthogonality

What did we have to change here? We added one line for each new feature, but were able to leave the old code untouched. This is a very elegant design: adding the new set of features does not change the underlying interpreter for the old language. This is called *orthogonality*.

Exercise: What is the price that programmers pay for the orthogonal design of SML?

8 Modeling Simple Control

8.1 The Semantics of Continuations

Here are some examples of the behavior of **letcc**:

1. (**letcc** *Xit* (+ (*Xit* 15) 5)) = 15

2. (**letcc** *Xit* (**lambda** (*x*) (+ (*Xit* (**lambda** (*x*) 5)) *x*))) cannot be evaluated any further; the outer (**lambda** ...) is a value, and *Xit* occurs free in it

3. The reduction sequence

```
      ((letcc Xit (lambda (x) (+ (Xit (lambda (x) 5)) x))) 25)
  ==> (letcc Xit2   ; since letcc "captures" its evaluation context
        ([lambda (x) (+ ((lambda (v) (Xit2 [v 25]))
                         (lambda (x) 5))
                        x)]
         25))
  ==> (letcc Xit2
        (+ ((lambda (v) (Xit2 [v 25])) (lambda (x) 5)) 25))
  ==> (letcc Xit2
        (+ (Xit2 [(lambda (x) 5) 25]) 25))
  ==> (letcc Xit2 (+ (Xit2 5) 25))
  ==> 5
```

In general, when a **letcc** expression is evaluated, it turns its current context (as in, "complete textual context") into a procedural object. This procedural object is also known as a *continuation object*. When a continuation object is applied, it forces the evaluator to remove the current evaluation context and to re-create the context of the original **letcc** expression, filled with the value of its argument. Like procedures, continuation objects are first-class values, which means they can be stored in data structures or tested by predicates.

Now we understand the semantics of **letcc**, but how do we use it to write programs? We need to also understand its pragmatics. For instance, **letcc** can be used for *exception handling*.

Example

Let us write a procedure, *Pi*, that computes the product of a list of digits. The data definition looks like this:

lod ::= null | (*cons* [*0-9*] *lod*)

Our procedure might look like this:

```
(define Pi
  (lambda (l)
    (cond
      ((null? l) 1)
      (else (* (car l) (Pi (cdr l)))))))
```

However, suppose it is possible that we can get an invalid digit (in the range [*a-f*]); if we do, we want the result of invoking *Pi* to be false. We can add the following clause within the **cond** statement,

```
((bad? (car l)) (Xit #f))
```

where *Xit* is some unused identifier.

We use the following recipe for constructing such programs:

- From the data description, recognize the exceptional data.
- In the corresponding code, add the call (*Xit V*) (where *V* represents the value to be returned in an exceptional situation) where *Xit* is some new identifier.
- Bind *Xit* by making it a parameter in the argument list to the procedure.
- Change all calls to that procedure to reflect the new arity. Since we don't know what *Xit*, we simply pass it along at all these call sites.
- Write a wrapper function to the desired procedure which takes the desired number of arguments. For instance,

```
(define Pi^e    ; Is this irrational/transcendental?
  (lambda (l)
    (letcc XXX (Pi l XXX))))
```

If we pass this new procedure exceptional data, we get

```
   (Pi^e '(1 2 b))
⟹ (letcc XXX (Pi '(1 2 b) XXX))
⟹ (letcc XXX (* 1 (* 2 (XXX #f))))
⟹ #f
```

as desired.

Exercise:

- Instead of creating a separate procedure, Pi^e, could we have written (**define** *Pi* (**letcc** *Xit* ...)) instead?
- Along similar lines, could we have written (**define** *Pi* (**lambda** ... (**letcc** *Xit* ...))) instead?
- Could we have hidden the code for *Pi* inside that of Pi^e using **letrec**?
- If we did this, would we still need to pass the continuation around?

8.2 Modeling Simple Control Constructs

There are numerous control constructs that we can add to LC. Some of these are:

- (*raise M*) stops computation and returns the value of *M*. (*raise M*) corresponds to an exceptional datum condition for the meta-evaluator. Hence, it can be added to the evaluator by following the steps above.
- Sometimes, we would like to control the power of an *raise* by delimiting the extent to which it can escape. Such a construct is called an *abort delimiter*, and is sometimes written as *prompt* or #.

  Suppose in Scheme we wrote

  ```
  (lambda (f G)
    (open-file f)
    (G f)
    (close-file f))
  ```

  If *G* executes an *raise* statement, then the file will never be closed. This might be undesirable. To prevent this, we can instead write

  ```
  (lambda (f G)
    (open-file f)
    (# (G f))
    (close-file f))
  ```

  # can be added to the evaluator with the following code:

  ```
  ((#? M) (letcc NewXit
             (MEval (#-body M) env NewXit)))
  ```

  This is a non-orthogonal change to the interpreter, but it is orthogonal with respect to the *language*, since the functional core remains the same.
- We could extend the abort delimiter to be of the form (# *M H*) where *H* is invoked only if *M* aborts. (The code in *H* might typically be used to perform some clean-up action.) Additional extensions are possible: we could have labeled exceptions, and we could also have restartable exceptions (where *raise* returns a value and the continuation active at the time it was invoked).

Here is the core of an interpreter that implements # and *raise*. This version of # takes a body and a handler, as outlined above. The handler takes one argument, which is the value "thrown" by *raise*.

```
(define MEval/ec
  (lambda (M env Exit)
    (cond
      ...
      ((raise? M)
        (Exit (MEval/ec (raise-expr M) env Exit)))
      ((#? M)
```

```
((letcc new-Exit
   (lambda ()
     (MEval/ec (#—body M) env
       (lambda (raised-value)
         (new-Exit (lambda ()
                     (MApply/ec
                       (MEval/ec (#—handler M) env Exit)
                       raised-value Exit)))))))))))
```

(Note that all calls to the former *MEval* will now have to call *MEval/ec* instead, passing on the *Exit* handler unchanged; only # installs new handlers. *MApply/ec* is similarly modified.)

8.3 Summary

At this point, we will conclude our study of meta-interpreters. We have thus far covered the following:

- interpretation of the mathematical expression language,
- destructive updates and mutable records, and
- simple control structures.

It would be worthwhile to note in passing some of the topics that we did not cover but which could be studied with the same methodology:

- Structuring language constructs
  - Modules
  - Objects
  - Capabilities
  - Abstract Data Types
- Parallel/distributed/concurrent programming
- Threads
- Intelligent backtracking

9 Eliminating Meta-Errors, and Self-Identifying Data

Understand thyself.
—Aristotle

Starting with this section, we will attempt to eliminate uses of meta-level interpretations. The behavior that an interpreter *specifies* should be as independent of the meta-language as possible. Independence guarantees a number of properties, most importantly a standard and portability of programs.

9.1 Errors

Consider what our interpreter for LC might do when we feed it an erroneous input such as

(*MEval* '(5 6))

Depending on which language we use to implement the interpreter, we will get different results. For instance, a Scheme implementation might report one of the following:

```
Error: attempt to apply non-procedure 5
apply: not a procedure (type was <fixnum>)
```

This is an example of *inheritance*, wherein the implemented language has inherited a behavior (in this case, for errors) from the implementing language. Another example of this behavior inheritance is the treatment of errors in C programs. Depending on the platform (operating system and hardware) that they run on, C programs behave differently when errors arise. A language specification should define when an error arises and, for clarity, what happens when an error occurs.

Consider the following fragment of the interpreter:

```
((numeral? M) (numeral-n M))
((add? M) (+ (MEval (add-lhs M) env)
             (MEval (add-rhs M) env)))
((lam? M) (make-closure M env))
((app? M) (Apply (MEval (app-fun M) env) (MEval (app-arg M) env)))
```

Where do we inherit error behavior from Scheme?

- Use of $+$. Instead of using the underlying addition operation directly, we can instead test for both arguments with *number?* and invoke it only if the tests succeed, signaling an error otherwise.

```
(define LC-+
  (lambda (m n)
    (if (and (number? m) (number? n))
      (+ m n)
      (LC-error ...))))
```

- Unguarded use of Scheme's application: we should instead check that it is a closure that is being applied.

```
(define Apply
```

```
(lambda (f a)
  (if (closure? f)
    (let ((body (closure-body f)))
      (MEval (lam-body body)
        (extend (closure-env f) (lam-arg body) a)))
    (LC-error ...))))
```

Suppose we were to add a conditional statement, **if**, to our language. Then we might have the following implementation:

```
((If? M)
 (if (MEval (If-test M) env)
     (MEval (If-then M) env)
     (MEval (If-else M) env)))
```

Unfortunately, this definition will not work: presently, all Scheme values returned by *MEval* will satisfy the test position of an **if** expression, so the failing branch will never be taken. Let us pick 0 to represent falsehood. Say we pick the numeral 0 for this purpose. Then we could rewrite the test as

```
(if (≠ (MEval (If-test M) env) 0)
   ...
   ...)
```

but in the process, we have introduced another inheritance of Scheme's error-handling, since we cannot be sure the value returned by *MEval* will always be a number. This is easily added.

Exercise: Why do some languages like Scheme and C pick one value to represent falsehood, and allow all others to represent truth?

Exercise: By introducing errors, we have introduced something into the language that we haven't explained using the LC interpreter itself. What is this?
Solution: We have asked questions such as *number?*.

9.2 Self-Identifying Data

Just as we introduced abstract syntax to sever our dependence on the meta-language to represent programs, we should similarly introduce meta-values so that we don't rely entirely on the meta-language to represent values. We can do this with declarations such as

```
(define-structure (Num n))
(define-structure (Clos param body env))
(define-structure (Bool b))
```

so now we can now write *Apply* with *Clos?*

```
(define Apply
  (lambda (f a)
    (if (Clos? f)
      ...)))
```

rather than with *procedure?*.

However, this is not an entirely satisfactory solution: since **define-structure** is a large and complex beast, we haven't really explained much by replacing, say, *number?* with *Num?*. Hence, we could instead use definitions like

```
(define make-Num
  (lambda (n)
    (list 'Num n)))
```

and then, adopting a convention that all data are now represented as lists with a tag at the head, define

```
(define Num?
  (lambda (LC-val)
    (eq? (car LC-val 'Num))))
```

Exercise: What other changes do we need to make, *eg*, in *LC-+*?

Let us now consider our definition of *Num?*. The symbol 'Num is easily represented as a number and *eq?* is a simple hardware instruction. We still need to explain *car*.

What we just discovered is that self-identifying data rely on the meta-language's ability to allocate data dynamically. This is also true of closures and arrays. Given the importance of this concept, we study an explanation of dynamic allocation.

10 Modeling Allocation Through State

We now explain how to implement the dynamic allocation procedures and their cognates. To avoid confusion, we shall replace 'c' with 'k' to yield *kar*, *kdr*, *kons* and friends. (Contrary to popular belief, this spelling is not the outcome of certain Teutonic influences.)

What does *kons* correspond to in C? It does two things: it dynamically allocates two elements of memory, and it initializes the elements with the values provided as arguments. In C, this roughly corresponds to *malloc* followed by initialization.

Exercise: Why do we need *malloc*? Can't the allocation be performed on the stack instead?

Moral: *Data have dynamic extent*.

All modern languages have the ability to create data whose extent is dynamic. However, many programming languages force the life-span of data to coincide with the execution period for the lexical scope in which they are created. As the "dangling pointer" problem of C and C++ programs illustrates, this brute-force approach to memory management leads to insidious, difficult-to-find bugs. Advanced languages instead put all complex data on the heap and let the program *behavior* determine the life-span of each datum. To make this approach work, such languages provide a garbage collector, which automatically de-allocates data when it is provably irrelevant.

Exercise: What is wrong with the phrase, "the lexical scope is exited"?

10.1 Modeling Allocation

Now we return to the question of how to model *kons* in the interpreter. We can do it by adding a *store* as a component of the evaluator, similar to the store in the Jam 2000 interpreter used in Comp 210. A store consists of two things: (1) a pointer to the next usable portion, and (2) an allocator that moves the next-use pointer and returns a handle on the allocated space.

Currently, our stores have the following abstract specification:

```
   setup: ()         -> ()
allocate: val x val -> loc
  lookup: loc       -> val
  update: loc x val -> ()
```

This can be implemented with the following code (which is written without error checks, to simplify presentation):

```
(define next-usable 0) ; of type 'loc'ation
(define memory '())

(define setup
  (lambda ()
```

```
    (set! memory (vector BIG))))

(define allocate
  (lambda (v-1 v-2)
    (begin0
      next-usable
      (vector-set! memory next-usable v-1)
      (vector-set! memory (+ next-usable 1) v-2)
      (set! next-usable (+ next-usable 2)))))

(define lookup
  (lambda (loc)
    (vector-ref memory loc)))

(define update
  (lambda (loc val)
    (vector-set! memory loc val)))
```

Given these, we can add *kons* to the interpreter:

```
((kons? M) (let ((l (MEval (kons-lhs M) env))
                 (r (MEval (kons-rhs M) env)))
             (make-LC-pair (allocate l r))))    ; DANGER!
```

(Recall that the value returned needs to be an *LC-pair*.) Why do we have a danger sign? It is because the order of evaluation of bindings in a **let** expression in Scheme is not specified. Hence, we may end up evaluating the *rhs* before we do the *lhs*. Since this is undesirable, we need to use a **let**∗ instead of **let**.

We can similarly add *kar*, *kdr*, *set-kar!* and *set-kdr!*:

```
((kar? M) (let ((p (MEval (kar-exp M) env)))
            (unless (LC-pair? p) (error ...))
            (lookup (LC-pair-loc p))))

((set-kdr!? M) (let* ((p (MEval (set-kdr!-exp M) env))
                      (v (MEval (set-kdr!-val M) env)))
                 (unless (LC-pair? p) (error ...))
                 (update (+ (LC-pair-loc p) 1) v)))
```

Exercise: Throughout this presentation, we have assumed the existence of mutation primitives in the underlying language to model the allocation and mutation primitives of LC. What might we do if our meta-language did not possess such primitives?

---

11 Modeling State Through Allocation

11.1 Store-Passing

The technique called *store-passing* arises from the following challenge: implement an interpreter for LC including *kons*, *kar*, *kdr*, *set-kar!* and *set-kdr!* using allocation but no side-effecting operations.

The technique we will use is to make the store a parameter to all the procedures, record changes in the store through functional update, and to return the updated store from any procedure that might change it. These changes are reflected in the following abstract specification for our store operations:

```
   setup: ()                -> store
allocate: store x val x val -> loc x store
  lookup: store x loc       -> val
  update: store x loc x val -> store
```

We now present an implementation of this specification, representing stores as a list of two elements: the next-use pointer and an association list of "mutations" made to the store.

```
(define make-store ; int × (int,value) list → store
  (lambda (next-use allocations)
    (list next-use allocations)))

(define next-use-of ; store ⟶ int
  (lambda (store)
    (car store)))

(define table-of ; store ⟶ (int,value) list
  (lambda (store)
    (cadr store)))

(define setup
  (lambda ()
    (make-store 0 '())))

(define allocate
  (lambda (s a b)
    (let ((first-free (next-use-of s)))
      (list first-free            ; location
        (make-store (+ first-free 2) ; store
          (cons (list (+ first-free 1) b)
            (cons (list first-free a)
              (table-of s))))))))
```

lookup extracts the association list and looks up the value corresponding to the given location in that list. Finally, the implementation of *update* is

```
(define update
  (lambda (s l v)
```

```
(make-store (next-use-of s)
  (cons (list l v)
    (table-of s)))))
```

In harmony with the above changes, we need to also change *MEval* to pass the store around. We will call this new evaluator *SMEval*. It needs to take an expression, an environment and a store, and return a value *and* a store. First, let us see how the evaluation of simple things like numerals changes (where *st* is the name for the store argument):

```
((numeral? M) (list (make-Num (numeral-n M)) st))
```

We have chosen to represent the two returned values — the result of evaluation, and the store — as a list. Other representations, such as Scheme's multiple-value mechanism, are also possible.

Now we consider a slightly more complex example, the primitive *plus*:

```
((plus? M) (let* ((L (SMEval (plus-lhs M) env st))
                  (R (SMEval (plus-rhs M) env (store-of L))))
             (list (add (value-of L) (value-of R)) (store-of R))))
```

Note how converting to store-passing style makes explicit the dependence on the order of evaluation. Finally, we need to adapt the implementation of the mutating primitives, such as *set-kar!*:

```
((set-kar!? M) (let* ((p (SMEval (set-kar!-pair M) st))
                      (v (SMEval (set-kar!-value M) (store-of p))))
                 (list your-favorite-value
                   (update (store-of v) (value-of p) (value-of v)))))
```

Hence, we have explained in a fairly different way what it means to add side-effects to the language; in particular, we have done so without relying on side-effects in the meta-language.

When the store is propagated in this manner, it is said to be *threaded*. The environment, in contrast, is not threaded; it is, in theory, duplicated across its various uses.

Exercise: What is the difference between the environment and the store?
Solution: The environment corresponds to the lexical scoping structure of the program, while the store captures the program's data creation behavior.

While converting an implementation to store-passing style requires an extensive re-write of the interpreter, there are also some benefits to be derived from using this style. These include:

- We have a model of imperative assignment which does not rely on assignment in the meta-language. This is particularly useful if the meta-language does not have assignment operations.

- Suppose, in the implementation of *set-kar!*, we had instead written

  ```
  (update (store-of p) (value-of p) (value-of v))
  ```

  in the last line. What would this do? It would create a new primitive that still changes the *kar* field of the indicated pair, but that does so without recognizing any of the side-effects that may have taken place in the process of determining the value being put into the pair. Put differently, an understanding of language constructs without an equivalent meta-language construct enables researchers to explore alternatives in programming language design.

- The natural lifetime ("dynamic extent") of a datum is unrelated to its lexical scope. This is illustrated by comparing the threaded store against the un-threaded environment.

Exercise: What do the environment and the store contain during and after the evaluation of

(**let** ((*x* (*malloc* ...)))
  body)

and what does this tell us about the relationship between lexical scope and dynamic extent?

11.2 Mutation *vs.* Allocation

The previous section and this one are duals. In the previous section, we assumed that our language had mutation operators, and showed how we may use these to model allocation (and mutation) in LC. This corresponds quite closely to the architecture of most modern computers, where a certain fixed amount of virtual memory is present at start-up, and allocation is modeled by apportioning fragments of this memory to individual processes upon request. Indeed, the implementation of *allocate* presented in the previous section is conceptually quite similar to that of Unix's *sbrk* primitive, which is used by C's *malloc* library routine.

In contrast, in this section, we have shown how to model mutation by assuming only the presence of dynamic allocation routines in our language. This is unlike the architectural model of most stock hardware. We found that under this assumption, we were required to make fairly extensive changes to our implementation of LC, but in return, we gained fine-grained control over the effect of mutations, and were able to understand better the distinction between static scope and dynamic extent.

12 What is a Type?

One of the central tenets of software engineering is that of early error discovery. Many programming languages support this idea with a type system. Roughly speaking, a *type* is a name for a collection of syntactic values, and a type system encourages the programmer to think about the type of the value that each phrase in the program produces.

Suppose the expression

(**if** *big, ugly expression*
  (5 6)
  a-nice-value)

is embedded deep in a program. If program tests never force the evaluation of *big, ugly expression* to return true, the error in the then branch is never discovered. However, it may still be the case that some input will eventually force the evaluation of (5 6), which will then generate a run-time error. This error could clearly have been avoided if the programmer and/or the language implementation had flagged the value 5 as something that is inappropriate for the function position of an application.

Idea 1 *Types are names for sets of syntactic values.*

In LC, we have two classes of syntactic values:

- integers: 5, 23, ...
- functions: (**lambda** (x) M), ...

Idea 2 *The valid sets of "input values" for each program operation can be described in terms of types.*

In LC, we have two program operations that compute values: addition and application. The former only accepts numbers, while the latter must receive a function value in the first position.

While this is not all that we would like to be able to say, this is all we *can* say in our current type framework. How might we wish to extend this? For instance, it would be useful to be able to specify what type of argument a procedure can accept. This would enable us to flag erroneous programs such as

((**lambda** (x) (x 10)) 5)

In the following fragment, what can we put in place of the $\diamond$?

((**lambda** (f) ($+$ (f 10) 5)) $\diamond$)

Since the argument gets applied, it must be a function; since its argument is an integer, it must be a function that accepts integers; and since its result is an argument to $+$, the result must also be an integer. Hence, $\diamond$ can be any function that accepts and returns integers.

From these two examples, we see that we would like to specify two things about the type of a function: its domain and its range. Syntactically, we will write it as $td \longrightarrow tr$ where td is the type of the domain, and tr is the type of the range. Hence, our grammar for types is

```
Type = 'int | Type '-> Type
```

where $'\longrightarrow$ is called a *type constructor*, since it builds a more complex type out of simpler ones. For example, $(int \longrightarrow int) \longrightarrow (int \longrightarrow int)$ is the type of the discrete difference operator.
NOTE: It is not meaningful to speak of *the* function type; indeed, there is an infinite number of function types.

Exercise: What is C's syntax for types? In particular, how does it represent function types?

12.1 Type Checking

Once we have names for sets of values, we can try to determine to which set the result of an expression belongs and see whether this result makes sense in the given context.

Idea 3 *Tell me what the types of the variables in an expression are and I'll tell you what the type of the expression is.*

In LC, as in every other language, we have free and bound variable occurrences. If we attach types to every binding occurrence of a variable, we have clearly covered all bound occurrences. This idea suggests a small modification to LC's syntax:

M ::= *var* | ('lambda *var type M*) | (*M M*) | *n* | (+ *M M*)

But what do we do about expressions with free variables? For those we must assume that we are given their types. The association of a free variable with a type is sometimes called a *type context* or a *type environment*. Given any expression and a type environment that covers all of its free variables, we can determine the type of the expression and can thus predict what kind of result it will produce.

Here is a type checker for LC, using a straightforward natural recursion formulation.

```
;; Type check a closed abstract representation of an LC expression
(define TypeCheck
  (lambda (are)
    (TC are (mt))))

;; Type check an open abstract representation of an LC expression
;; are: an abstract LC expression
;; tenv: an environment that associates variables with types
;; result: the type of are in tenv
;; effect: error if the types don't work out
(define TC
  (lambda (are tenv)
    (cond
      ((var? are) (lookup (var-name are)
                    tenv
                    (lambda () (error 'TC "free var: ~s" are))))
      ((const? are) 'int)
      ((add? are)
```

```
        (Type= (TC (add-left are) tenv) 'int (add-left are))
        (Type= (TC (add-right are) tenv) 'int (add-right are)))
      ((if? are)
        (Type= (TC (if-tst are) tenv) 'int (if-tst are))
        (Type= (TC (if-thn are) tenv) (TC (if-els are) tenv) are))
      ((app? are)
        (let ((funT (TC (app-rator are) tenv)))
          (unless (->? funT)
            (error 'TC "function type expected for ~s~n  ~s inferred~n"
              (app-rator are) funT))
          (Type= (->-domain funT)
            (argT (TC (app-rand are) tenv))
            (app-rand are))
          (->-range funT)))
      ((proc? are)
        (let ((ptype (proc-type are)))
          (make--> ptype
            (TC (proc-body are)
              (extend tenv (proc-param are) ptype)))))
      ((rec? are)
        (let* ((var-type (rec-type are))
               (rtenv (extend tenv (rec-var are) var-type)))
          (Type= (TC (rec-rhs are) rtenv) var-type (rec-rhs are))
          (TC (rec-body are) rtenv)))
      (else (error 'TC "impossible: ~s" are)))))

;; Compare two types, raise error if mismatched
;; recd: the type that was inferred for b
;; expected: the expected type (according to context)
;; result: recd if recd and expected are structurally identical types
;; effect: call to error with are if the types don't match
(define Type=
  (lambda (recd expected b)
    (if (equal? recd expected) recd
      (error 'Type= "expected: ~s; constructed: ~s~n   for ~s"
        expected recd b))))
```

Using the type checker we can formulate the following more concrete claim about LC (which is basically a formal version of Idea 3):

Idea 4 *If (TC M mt-env) = t and if (Eval M mt-env) = V, then (TC V mt-env) = t.*

NOTE: LC satisfies this but most so-called "strongly typed" languages don't. In C programs, for example, even a variable declared of some type isn't guaranteed to contain a C value of that kind. This property is only true if we accept that C values are bit strings. Pascal and Ada have similar problems.

12.2 Typing Rules

In the previous section, we have formulated a type-checker as a program. It is possible to describe a type theory in a more language-independent fashion, and indeed, most modern presentations of type theories use this style. The basic idea of this form of presentation is to establish *type judgments* of the form

```
tenv |- M : type
```

with a formal proof system. In this judgment, *tenv* is a type environment; the 'turn stile' (`|-`) is read "proves"; *M* is a term; the colon (`:`) reads as "has the type"; and the *emphasized* text is the derived type. We then have the following rules:

- We begin with an *axiom* that tells us we can extract information reposited in the environment:

```
tenv |- x : t    if tenv(x) = t
```

- Next, we have an *inference rule* that shows us how we construct function types; this is where type assertions are introduced into the type environment:

```
      tenv + [x : t] |- M : s
-----------------------------------
tenv |- (lambda (x : t) M) : t -> s
```

- Finally, we show the typing of applications:

```
tenv |- F : t -> s    tenv |- A : t
-----------------------------------
         tenv |- (F A) : s
```

Hence, inference rules produce *proof trees*. The text above the lines (called the *antecedents*) are judgments that need to be established using inference rules and axioms. When we use only axioms, we have finished our job. Notice that the rule for application requires two proofs to establish the antecedent, while the one for procedures requires only one.

13 Types and Safety

We will henceforth refer to our typed version of LC as TLC. TLC is an extremely "boring" programming language for the following reason: every program in TLC terminates. For example, consider the canonical infinite loop

((**lambda** *x* (*x x*)) (**lambda** *x* (*x x*)))

which we write in TLC as

((**lambda** *x t* (*x x*)) . . .)

where t is the type we assign the parameter. Since x is applied in the body of the procedure, it must have some type $t = u \longrightarrow v$. But the argument to x is x itself, so the type u must be whatever t is. Hence, we need a type $t = t \longrightarrow v$ to be able to type this application. You should be able to convince yourself that we cannot construct such a type from our inductive definition of types for TLC.

We cannot type this infinite loop. Indeed, a stronger property holds: *Every program in TLC terminates*. The following theorem holds for TLC: *For all programs M, if M is of type t, then M evaluates to a value of type t*. Since programs are just closed expressions, a similar theorem holds for all (closed) sub-expressions of programs.

Just as we cannot express infinite loops, we also cannot introduce other constructions such as pairing without explicitly adding type judgments for them to the language.

Thus, we see that types have imposed a tremendous restriction on our ability to express computations. Of course, this *is* the purpose of type systems: to restrict the expressivity of a language so that programmers cannot easily shoot themselves in the foot.

Let us consider some extensions we can add to TLC and examine their associated typing rules.

13.1 Explicit Recursion

We extend TLC with terms of the form

(*rec-let x tx Vx B*)

that have the following type judgment:

```
tenv + [x : tx] |- Vx : tx    tenv + [x : tx] |- B : t
-----------------------------------------------------
           tenv |- (reclet x tx Vx B) : t
```

13.2 Pairs

We can add the constructor *cons*, which has type `t s -> t x s` (read: "*cons* takes two arguments, of type *t* and *s* respectively, and returns an object of which is a Cartesian product, whose first projection has type *t* and the second *s*"). Note that juxtaposition is used for arguments to a function, while the *x* operator represents a type constructed from two component types.

In adding *cons*, we have added a new constructor to our language of types:

t ::= . . .
    | (*t x t*)

We can then define *car* and *cdr* to represent the projection functions:

```
|- car : t x s -> t
|- cdr : t x s -> s
```

However, note that any argument list must still be a tuple whose length is pre-determined and hence cannot be arbitrary. Thus, this *cons* cannot be used to create lists.

13.3 Lists

To circumvent this, we can add lists explicitly to the language. First, we must again extend our type syntax to represent the list type. For simplicity, we shall assume that lists can only contain integers. Then the type of an integer list is *ilist*:

$t ::= \ldots \mid ilist$

We then specify the types of the following constants and primitives:

```
null : ilist

cons : int ilist -> ilist

car : ilist -> int
cdr : ilist -> ilist
null? : ilist -> bool
cons? : ilist -> bool
```

We should consider whether these additions to TLC change the property we stated earlier, *ie*, that all TLC programs terminate. If we add explicit recursion to our language, we can write programs such as the following:

(*reclet f* (*int* $\longrightarrow$ *int*)
(**lambda** *x int* (*f x*))
(*f* 0))

This program will not terminate, which is contrary to the original statement. Now the Central Theorem of Typed Languages is: *For all programs, if M is of type t and if M evaluates to a value V, then V has the type t.*

The addition of pairs to TLC does not change the above statement of the Central Theorem of Typed Languages. However, consider the addition of lists. What happens if we evaluate the expression (*car null*)? According to the typing rules, *null* is of type *ilist*, so the application is well-typed; however, the result of this expression cannot be any meaningful integer. What should the result be?

This leads us to the following dictum: *Types are Lies*.

13.4 Safe Implementations

There are several approaches we can take to resolving this situation. Some of these are:

1. To preserve the theorem, we can define that operations such as (*car null*) and (/ 1 0) diverge.

2. Restate the claim as *Typed programs never cause run-time errors; they raise one of the exceptions that are explicitly listed in the specification of the semantics.*

The latter solution is preferable to the former one for two reasons: (1) it provides the programmer with useful information about errors, and (2) it closely corresponds to the practice of a large class of language implementations, including Java and ML. We call such language implementations *safe*.

In a safe implementation, the behavior upon encountering each error is clearly specified. In contrast, an unsafe implementation would depend upon the meta-level error processing. For instance, in C, if an element beyond the end of an array is accessed, the behavior is unspecified: it might yield an error (at the meta-level), or it might blithely return some value. In contrast, a safe language like ML clearly specifies what action to take: for instance, it might contain rules such as

```
(car null) : int ==> (raise IListEmpty) : int
```

What we see here is that types alone do not guarantee anything about a language. We also need to have a safe implementation of the programming language to enjoy the guarantees promised by a Type Theorem. However, even if a language is typed and safe, a programmer must think about those phrases in his programs that may raise pre-defined run-time exceptions (such as (*car null*) or (/ 1 0)). It is the goal of MrSpidey [4], a component of DrScheme [3], to help the programmer reason about such problems, especially the use of *car* or *cdr* on empty lists.

14 Types and Datatypes

Types are the thought-police of the programming language world.

In the first part of this course, we discussed how to model the traditional class of language facilities with interpreters. In the second part, we are studying how to understand typed languages. Remember that types impose syntactic restrictions on programmers to prevent simple mistakes. Typically these mistakes re-inforce the abstraction boundaries a programmer thinks about as he programs, *eg*, that some variable ranges over integer values and another over integer functions. In practice, we will restrict programmers by making certain program phrases illegal, and not accepting such phrases.

Typed languages require two extensions:

1. A sub-language for talking about types.
2. A mechanism for stating what the types of certain primitive phrases are, and rules for inferring the rules for the remaining phrases in a program. In our study we have followed the usual strategy of specifying the types for all binding occurrences of of identifiers and of inferring the types of the other phrases in a program with a type checker.

Next we take a closer look at what kind of restrictions our simple type system imposed on TLC, and how to get this expressive power of programming back.

14.1 Types are Restrictive

Consider the Scheme procedure *assq*, which takes two inputs, *x* of type *sym* and *list* of type (*sym value*) *list*. It returns a value (*x*,*v*) if *x* is associated with *v* in *l*, or #f otherwise. We cannot write *assq* in TLC; the essence of the problem can be described in the following expression:

```
(if P
  #t
  0)
```

where *P* is some expression whose type is *bool*. This expression cannot be typed in TLC because both branches of a conditional expression in TLC have to have the same type. Likewise, we cannot use lists with *assq* unless all the elements have the same type. Both restrictions are severe considering the flexibility of Scheme's *assq*.

We could avoid these problems if we added a new kind of type constructor to our type language:

$t ::= \ldots \mid (t + t)$

which is known as a *union type*. A type ($t1 + t2$) indicates that the corresponding expressions have either type *t1* or type *t2*.

Now consider the following program, which sums up the leaves of a binary tree:

```
(define Sigma
  (lambda (tea : tree)
    (if (is-a-leaf? tea)
      tea
      (+ (Sigma (left tea)) (Sigma (right tea))))))
```

Can this program be typed in TLC?

Clearly, *tea* is of type *tree*; but we want to view it as a number or as a combination of two trees. In Scheme, we would write a data description that looked like this:

```
A Binary Tree is either
  a number, or
  a pair of Binary Trees.
```

But with the current type system, we cannot write down such a description for arbitrarily large binary trees; we can only construct trees of fixed, finite depth.

What we really need is the ability to make a component of a type refer to a part of itself, *ie*, a *recursive type*. In C, we would solve this problem by using pointers, but this solution is extremely low-level. *Pointers are to data structures what labels and goto's are for program structures*: they are all low-level methods that make it easy for the programmer to make mistakes. Indeed, pointers are typically worse than goto's: few languages provide ways to jump to computed labels, but many languages, especially C, provide ways to dereference computed pointers.

A **datatype** declaration has the following format:

(**datatype** *tid*
  ((*constr-1 type-11 type-12* ...)
   ...
   (*constr-n type-n1 type-n2* ...))
 exp)

This declaration creates a new type whose name is *tid*. The type can have several forms, called *variants*, which are described by the entries in the declaration. Each variant has a *constructor*, whose name is the first field of that variant's entry. Each *n*-place constructor is followed in the entry by *n* types, which specify the types of the arguments allowed for the constructor. The **datatype** declaration provides a means for declaring union types; most crucially, the scope of *tid* is the entire **datatype** declaration, so recursive types can also be created in this manner.

For example, consider the declaration of the type of binary trees, of the sort we summed above:

(**datatype** *BT*
  ((*leaf int*)
   (*node BT BT*))
 exp)

However, if we try writing code that uses this type declaration, we soon find that we are missing some key items, such as a means for telling the different variants apart (*predicates*) and means for getting at the individual fields in a component (*selectors*). Hence, we extend the syntax of a **datatype** declaration to include these:

(**datatype** *tid*
  ((*constr-1 pred-1* (*type-11 sel-11*) (*type-12 sel-12*) ...)
   ...
   (*constr-n pred-n* (*type-n1 sel-n1*) (*type-n2 sel-n2*) ...))
 exp)

datatype combines union types with recursive types. After all, the latter only make sense with the former, and the former can easily be introduced with **datatype**. For example, one might write the return type for *assq* as

(**datatype** *assoc-type*
  ((*SOME int*)
   (*NONE*))
 ...)

Exercise: Every Scheme program can be prefixed with a single **datatype** declaration, and then be explicitly typed in terms of that declaration. What would this **datatype** look like?

14.2 Typechecking Datatypes

We have thus far appealed to intuition to understand the typing properties of **datatype**. This intuitive understanding is not sufficient to answer questions such as, "How many types are created if we place a **datatype** statement inside a loop?" Hence, we shall construct a typing rule for **datatype**.

Clearly, the type of the (**datatype** ...) expression should be that of the result expression contained therein. So we can write

```
     ... tenv' |- exp : t
------------------------------
tenv |- (datatype ... exp) : t
```

where *tenv′* is possibly an augmented version of *tenv*. However, it would be quite useless if the type environments *tenv* and *tenv′* were the same, since the declarations would then have had no effect at all. In fact, the declarations augment the type environment in the expected manner. Each of the constructors, selectors and predicates is typed —

```
constr-1 : type-11 ... type-1m -> tid
pred-1 : tid -> bool
sel-11 : tid -> type-11
...
```

— and these types are then incorporated into *tenv′*, shadowing type bindings in *tenv* as appropriate:

```
tenv + [constr-1 : type-11 ... type-1m -> tid,
        pred-1 : tid -> bool,
        sel-11 : tid -> type-11,
        constr-2 : type-21 ... type-2k -> tid,
        ... ] |- exp : t
----------------------------------------------
        tenv |- (datatype ... exp) : t
```

This seems sufficient: we augment the type environment with the types of the procedures created by the datatype declaration, and then, using their types, type the contained expression, returning its type as that of the overall expression. What is wrong with this judgment?

One reasonable question is, "What if the value returned were of type *tid*?" It would not be a very meaningful value, since *tid*'s extent is the lexical (**datatype** ...) body; not only would there be no way of inspecting or using the returned value, it would not even have a meaningful type. Thus, we should prevent against this possibility. (Incidentally, the naïve approach is the one taken by ML.)

We therefore impose the restriction that *tid* cannot be free in the type of the expression portion of the **datatype** declaration. However, this is unfortunate, since there are, in fact, situations where it is meaningful to return an object of type *tid* along with some procedures that operate on such an object. This cannot be done with our typing restrictions. The types in such a program would, per force, have to be anticipated, and the declaration would have to be moved into a sufficiently global scope such that the creation and all uses of objects of that type are encompassed.

We are *still* not done! (This example should illustrate the extremely subtle problem of designing type systems.) Here is a "poem" that illustrates one remaining problem:

```
(datatype A
  ((cons-1 (int sel-1)))
 (datatype ⋄
   ((cons-2 (bool sel-2)))
  (if (sel-2 (cons-1 5)) ... ...)))
```

Depending on what $\diamond$ is, we may or may not notice the problem. We do not if it were, say, *B*; but we definitely do if it is *A*.

Exercise: Explain what the problem is, and show how it can be resolved.

---

15 Polymorphism

In the previous section, we introduced two key concepts: union types and recursive types. We also introduced the **datatype** mechanism, which can express both these concepts. With **datatype**, we can do away with some of the types we had introduced into TLC earlier, such as *ilist*, which can be written (using a more compact syntax) as

```
datatype ilist = n () + c ( int , x )
```

Exercise: Determine what these declarations represent:

```
datatype x = n () + c ( int , x )
in
  datatype y = ni () + ci ( int -> int , y )
  in
    ...
    datatype z = nz ( int ) + cz ( int -> z )
    ...
```

15.1 Explicit Polymorphism

Since we have arbitrary length lists available, we can write a procedure that maps its argument over a list of integers, returning a list of characters:

(**define** *map*
(**lambda** (*f* (*int* $\longrightarrow$ *char*) *l* (*list int*))
(**if** ($null_{int}$? *l*)
$null_{char}$
($cons_{char}$ (*f* (car_{int} *l*)) (*map f* (cdr_{int} *l*))))))

while we can also write a procedure that maps over a list of functions ...

(**define** *map*
(**lambda** (*f* ((*int* $\longrightarrow$ *int*) $\longrightarrow$ *int*) *l* (*list* (*int* $\longrightarrow$ *int*)))
(**if** ($null_{int \longrightarrow int}$? *l*)
$null_{int}$
($cons_{int}$ (*f* ($car_{int \longrightarrow int}$ *l*)) (*map f* ($cdr_{int \longrightarrow int}$ *l*))))))

Of course, these two procedures are identical but for the types specified for the arguments. In Scheme, we have only one *map* procedure that subsumes all of these; in languages like Pascal or TLC, we need to have one for each argument type.

If we look carefully at these declarations, we notice that only two of the types actually matter: the type of the elements in the list argument and the return type of the function argument. Everything else in the declaration is implied by these two parameters. Hence, we could rewrite *map* in this manner:

(**define** *map*

```
(lambda (α)
  (lambda (β)
    (reclet g ((β ⟶ α) ⟶ (list β) ⟶ (list α))
      (lambda (f (β ⟶ α))
        (lambda (l (list β))
          (if (null_β? l)
            null_α
            (cons_α (f (car_β l))
              ((g f) (cdr_β l))))))
      g))))
```

In this declaration, the arguments being passed to *map* are *types*; hence, we cannot write this procedure in TLC. So we need another extension to the type language to handle these kinds of abstractions. We call this new language XPolyLC.

1. We can add a new form of abstraction. We denote it with the keyword Λ, which is similar to **lambda** except that it binds *types*, not values, to identifiers. Using Λ, we can write *map* as

```
(Λ (α Omega)
  (Λ (β Omega)
    (reclet g ((β ⟶ α) ⟶ (list β) ⟶ (list α))
      (lambda (f (β ⟶ α))
        (lambda (l (list β))
          (if (null_β? l)
            null_α
            (cons_α (f (car_β l))
              ((g f) (cdr_β l))))))))
```

Recall that in TLC, we have to annotate each binding occurrence with its type. But what type can we assign to types themselves? We use *Omega*, which represents the set of all types. Since the "type" of all arguments bound by Λ will always be *Omega*, we shall henceforth leave this declaration implicit. This is the *introduction* rule for type abstractions.

Now consider this definition of *map*:

```
(define map
  (Λ (α)
    (Λ (β)
      (lambda (f (β ⟶ α))
        (lambda (l (list β))
          (if (null_β? l)
            null_α
            (cons_α (f (car_β l))
              ((map f) (cdr_β l)))))))))
```

When we perform the recursive call to *map*, we need to pass to pass the types first to instantiate *map*. However, we do not have any facilities for passing types as arguments, since they are not values.

2. Hence, we introduce an *elimination* rule for type abstractions. This is done in the form of a new kind of application, *Tapp* (for "type application"). *Tapp* invokes an objected created with Λ with a type as an argument. Hence, we would write the recursive call to *map* above as

$$(((Tapp\ (Tapp\ map\ \alpha)\ \beta)\ f)\ (cdr_\beta\ l))$$

with the rest of the code remaining unchanged.

In TLC, and indeed in many programming languages, a programmer provides type annotations at binding instances of variables, but leaves the type of individual expressions to be inferred (computed) and checked by the language implementation. In XPolyLC, we have added two new kinds of expressions, type introductions and eliminations, so we examine what the types of these expressions are.

1. What is the type of the (**lambda** (l (*list* β)) ...) expression in either definition of *map*?

   ((*list* β) $\longrightarrow$ (*list* α)).

2. What is the type of the (**lambda** (f ($\beta \longrightarrow \alpha$)) ...) expression in either definition of *map*?

   (($\beta \longrightarrow \alpha$) $\longrightarrow$ type of (1)).

3. What is the type of (Λ (β) ...) in the latter definition of *map*?

   The type is ($\beta \longrightarrow$ type of (2)), for *all* possible values of β. Hence, we write this as ($\forall\ \beta$. (2)).

4. Finally, what is the type of (Λ (α) ...)?

   This has type ($\alpha \longrightarrow$ type of (3)), for all possible values of α. We write this type as ($\forall\ \alpha$. (3)).

In reality, we need to perform a type application to instantiate each primitive as well. The resulting code looks like

```
(define map
  (Λ (α)
    (Λ (β)
      (lambda (f (β ⟶ α))
        (lambda (l (list β))
          (if ([Tapp null? β] l)
            [Tapp null α]
            ([Tapp cons α] (f ([Tapp car β] l))
              (([Tapp [Tapp map α] β] f) ([Tapp cdr β] l)))))))))
```

Hence, α and β range (abstract) over types in the same manner that **lambda**-bound identifiers range (abstract) over values. This property of a type system, wherein "type variables" are used to abstract over types, is called *polymorphism*. This new form of types is called a *type schema*, since it represents a framework which can be instantiated with several different types. This kind of type system is said to be *explicitly polymorphic*, since the programmer is required to manually specify each instance of type abstraction, and to perform the appropriate type applications.

15.2 Implicit Polymorphism

While explicit polymorphism adds considerable power to a programming language, it is not a comfortable system to program in, due to the need to keep track of the abstractions at the type level in addition to those at the value level. To counter this problem, we allow the programmer the following freedom: *every type declaration required in TLC can be left blank*. We then design typing rules that will attempt to *guess* an appropriate type such that the resulting expression will pass the type checker.

Consider the following example. If we have

(**lambda** (x ⋄) x)

what can we write in the blank? We could certainly use *int*; likewise, we could use *bool*. In fact, any well-formed type we write in the blank will satisfy the type checker.

Now consider a typing judgment for **let**:

```
Tenv |- Exp : t    Tenv + [f : t] |- Bexp : type
------------------------------------------------
       Tenv |- (let f t Exp Bexp) : type
```

Say the *Exp* is (**lambda** (x ⋄) x). Then how many ways are there to type it and use it in the body? We can only use whatever was put in the place of ⋄, so the resulting identity function can only be used on objects of the one type t above. For instance, it could not be used to type

(**let** ((*id* (**lambda** (x ⋄) x)))
  (**if** (*id true*)
    (*id* 5)
    (*id* 6)))

since the closure is applied to both booleans and integers. Hence, this is not a convenient typing rule. Instead, the SML programming language uses a modified judgment for **let**:

```
  Tenv |- Bexp [f / Exp] : type
---------------------------------
Tenv |- (let f _ Exp Bexp) : type
```

This rule literally copies the *Exp* expression through *Bexp*; when applied to the code fragment we examined earlier, we end up typing the expression

(**if** ((**lambda** (x ⋄) x) *true*)
  ((**lambda** (x ⋄) x) 5)
  ((**lambda** (x ⋄) x) 6))

As can be seen, everything — including the programmer's demand to guess a type — is copied by this rule. Hence, an independent guess can be made at each location.

The last rule is easy to implement, though it can be expensive in terms of the time taken for the typing phase. But, how do we implement guessing, especially since there is an unbounded number of types? The answer is that we *create* a new type, assuming no properties for it, and based on the constraints derived from typing the body M, we derive the "guessed" type.

In summary, polymorphism is the ability to abstract over types. *Implicit polymorphism* is the ability of a type system to assign many different types to the same phrase without the need for explicit type declarations. Several languages, including SML, Haskell and Clean, provide type inference ("guessing") with implicit polymorphism for **let** (and possibly other binding constructs) [9].

Exercise: What is the connexion between explicit and implicit polymorphism?

---

16 Implicit Polymorphism

In the previous section, we introduced polymorphism and distinguished between its explicit and implicit forms. When confronted with the former, the programmer writes down the types of all variables and explicitly abstracts over both values and types. For the latter, the programmer simply omits all type information from the program and tells the language implementation to derive the types. However, we left this process of derivation unspecified.

Consider the following examples:

1. (**lambda** (x $\diamond$) x): we can use *any* type in the type language in place of $\diamond$.

2. (**lambda** (x $\diamond$) ($+$ x 0)): we can only put in *int*, since the argument is used in an addition.

3. (**lambda** (x $\diamond$) (x 0)): since the argument is applied, we know its type must be of the form $a \longrightarrow b$. The type accepted by the argument, a, must be a number; its result type can still be any legal type. Hence, we can assign the type ($int \longrightarrow \diamond$) for this expression.

So it is still true that (**lambda** (x $\diamond$) (x x)) cannot be typed unless we introduce a datatype, since we are faced with the restriction that any type we write down must be expressible in the type language. In general, what can we do if we are given an expression of the form (**lambda** (x $\diamond$) <*body*>)?

In the above fragment, $\diamond$ represents an unknown type. We shall type such expressions in two stages:

1. We shall name and conquer: first, we assign it a type variable (hearkening back to explicit polymorphism). Thus, we introduce *type variables* into the language. Every variable has a type in this language:

   ```
   t_alpha = tv | int | (t_alpha -> t_alpha)
   ```

   where the *tv* are Greek alphabet representing type variables.

2. Once we have variables, we need to write down equations or constraints and then solve these. Where can we get these equations from? They arise naturally from the typing rules.

Examples

First, we show the typing of (**lambda** (x $\diamond$) x). Initially, we infer a type for the procedure based on two fresh type variables:

```
T |- (lambda (x alpha) x) : alpha -> beta
```

Now we type the body of the procedure. Since x is bound in the body, we have

```
T [ x/alpha ] |- x : alpha
```

However, since the type of the result of the procedure is that of x, and this result type has been indicated as β, we also have

```
T [ x/alpha ] |- x : beta
```

Combining these facts, we see that the type of the procedure is $\alpha \longrightarrow \beta$ subject to the constraint that $\alpha = \beta$, *ie*, $\alpha \longrightarrow \alpha$. Put differently, the programmer could have written down any arbitrary TLC type, and the program would have typed correctly.

This example illustrates the process of deriving a type. We use the type language *t_alpha* to play the guessing game, but the types given to the programmer are still in terms of the old types we had.

We now consider a more involved case:

```
((lambda (x _) x) (lambda (y _) y))
```

We begin by assuming a new type for the whole expression:

```
T |- ((lambda (x _) x) (lambda (x _) x)) : gamma
```

Now we type the sub-terms. Each one is a procedure, so each is given an "arrow" type.

```
T |- (lambda (x alpha) x) : delta -> epsilon
T |- (lambda (y beta) y) : phi -> psi
```

From this, we derive two equations:

```
gamma = epsilon
delta = phi -> psi
```

The former holds since the type of the overall expression is that returned by the applied procedure, the latter by the typing of arguments to procedures.

Next, we can conclude that

```
T [ x/alpha ] |- x : alpha
T [ y/beta ] |- y : beta
```

Each of these induces several more equations:

1. $\alpha =$ *delta* since both represent the argument,
2. $\alpha =$ *epsilon* since both represent the result;
3. $\beta =$ *phi* and
4. $\beta =$ *psi* similarly.

From these, we can in turn conclude that

```
delta = phi -> psi = beta -> beta
epsilon = alpha = delta = beta -> beta
gamma = epsilon = beta -> beta
```

which gives the type $\beta \longrightarrow \beta$ for the entire expression, where β is the type assigned to the argument of the procedure passed in as the argument.

The process of solving symbolic constraints is the extension of Gaussian elimination to terms over free (uninterpreted) algebras. It is called *unification* and was invented by Robinson at Rice (in the Philosophy department) in the mid-60s.

In spite of all this machinery, expressions like

```
(let ((f (lambda (x _) x)))
  (if (f true)
    (f 5)
    (f 6)))
```

still cannot be typed. The problem is that the rule we have adopted for **let** requires us to make a guess before we check the body; no matter which type we choose, we will run afoul. So we propose that, instead, we use the copying rule for **let** that was proposed in the previous section. What is the property of this rule that enables us to type this expression? It is that it *copies the blanks*. The "copying" **let** typing rule copies places for guessing types.

However, this new rule has one very deleterious effect. There could be a **let** in the binding for f; there could be another **let** in *its* binding; and so forth. When we copy the code, this leads to an exponential blow-up in the resulting (copied) code that needs to be typed. Thus, in practice, we want this typing rule, but would like to avoid having to duplicate code. In effect, this means that we want to put the $\diamond$'s in the *environment*, *ie*, we want to bind type variables at **let**s and free them at uses.

Idea 5 *Introduce type schemas with close and open operations.*

Our type schemas will take the following shapes:

```
ts = t_alpha | (forall (alpha ... beta) t_alpha)
```

In the above expression, we would get the type $\alpha \longrightarrow \alpha$ for f, which is $(\forall\ (\alpha)\ (\alpha \longrightarrow \alpha))$. This also shows that type inference for implicit polymorphism can be thought of as a translation process from TLC syntax without types into XPolyLC.

In the following example, we quantify over multiple type variables:

(**let** ((*k* (**lambda** (x α)
          (**lambda** (y β)
            x))))
  …)

The shape of type inferred for k is then $(\forall\ (\alpha\ \beta)\ (\alpha \longrightarrow (\beta \longrightarrow \alpha)))$.

With type schemas in hand, we try to produce a better **let** rule.

```
T |- E : t    T[ x/CLOSE(t) ] |- B : s
--------------------------------------
        T |- (let (x E) B) : s
```

When we look up identifiers in the type environment, we might sometimes get a *CLOSE*d object instead of a regular type. Hence, we need another typing rule, *OPEN*, to cover these cases:

```
T |- x : t [ alpha.../beta... ]
```

This rule is read as follows. Say an identifier x is looked up in the type environment T, and it has the type t. If t is a type schema, inserted into the environment through a *CLOSE* expression, then we replace the type variables quantified over in the schema (the *alpha...*) with *fresh* type variables (the *beta...*), then proceed as before. Since a new set of type variables is instantiated at each use, the same code can be given different types in different uses, so our canonical example can be typed.

Unfortunately, these simple versions of *CLOSE* and *OPEN* are flawed. Suppose we had this program:

(**lambda** (y $\diamond$)
  (**let** ((f (**lambda** (x $\diamond$) y)))
    (**if** (f *true*)
      (+ (f *true*) 5)
      6)))

Say y is assigned type α and x is assigned β. Then the type of f is $(\alpha \longrightarrow \beta)$. Our typing rule has naïvely closed over *all* the type variables lexically visible. At each use of f, new type variables are created. In the first, $beta\_1 = bool$ and $alpha\_1 = bool$; in the second, $alpha\_2 = int$ and $beta\_2 = bool$. The type checker is satisfied, and an expression that "type-faults" at run-time has passed the type checker.

In the rule for **let**, if we restrict *CLOSE* to only close over variables introduced by the polymorphic **let**, then we get the desired behavior. The introduction rule is now

```
T |- E : t    T[ x/CLOSE(t,T) ] |- B : s
----------------------------------------
         T |- (let (x E) B) : s
```

with the same elimination rule, and we can formally write down a specification of *CLOSE*:

```
CLOSE (t,T) = (forall (alpha...) t)
```

where $alpha... = ftv(t) - ftv(T)$ (*ftv* computes the set of free type variables in expressions and type environments). With this new typing rule, the type schema for f is $(\forall\ (\beta)\ (\beta \longrightarrow \alpha))$, so we now have to unify α with both *int* and *bool*, which fails and results in an error.

17 Types: Three Final Words

There are three more topics that we will consider before concluding our coverage of types.

17.1 Mutable Records

We already had mutable records in LC:

M ::= (ref M) | ($!$ M) | (M := M) | ...

Adding these to TLC requires typing rules.

First, we extend the type language:

t ::= ($tref$ t) | ...

Given the *tref* type constructor, the rest is straightforward.

```
      tenv |- M : t
--------------------------
tenv |- (ref M) : (tref t)

tenv |- M : (tref t)
--------------------
 tenv |- (! M) : t

tenv |- M1 : (tref t)    tenv |- M2 : t
---------------------------------------
       tenv |- (M1 := M2) : int
```

Since we don't care about the result of an assignment statement, we assume it will always return the integer 13.

This gives us mutation in the core of TLC. However, we made three significant extensions to TLC's type structure along the way: **datatype**, explicit polymorphism and implicit polymorphism. How does mutation behave in the presence of these extensions?

1. Adding references to **datatype** does not cause any difficulties. We gain the ability to create records with mutable fields.

2. References also integrate well with explicit polymorphism. For instance, we could have a type abstraction like

(Λ (α)
  (**lambda** (x ($tref$ α))
    ...))

and the combined typing rules suffice to type this.

3. Now assume we have implicit polymorphism in our language. Recall we have type judgments such as

```
 Tenv |- Bexp [ x/Exp ] : t
----------------------------
Tenv |- (let x Exp Bexp) : t
```

Exercise: What goes wrong if *x* does not occur free in *Bexp*?

Now consider the addition of mutable records. *ref* is a function; how do we type it? It takes one argument, say α, and returns an object of type (*tref* α). *!* is a projection, while *:=* takes a (*tref* α) and an α, and returns an *int*.

```
ref : (forall (alpha) (alpha -> (tref alpha)))
!   : (forall (alpha) ((tref alpha) -> alpha))
:=  : (forall (alpha) ((tref alpha) alpha -> int))
```

Since we are in an implicitly polymorphic language, we can omit type information at bindings, as in

(**let** (*f* (*ref* (**lambda** (*x*) *x*)))
  (**begin**
    (*:=* *f* (**lambda** (*x*) 5))
    (**if** ((*!* *f*) *true*)
      5
      6)))

In Scheme, this program would return 5. In TLC, the copying rule for **let** "expands" this into

(**begin**
  (*:=* (*ref* (**lambda** (*x*) *x*)) (**lambda** (*x*) 5))
  (**if** ((*!* (*ref* (**lambda** (*x*) *x*))) *true*)
    5
    6))

which passes the type-checker. However, an evaluation in TLC goes wrong. Instead of a boolean value, an integer will appear in the test position of the conditional during execution. If the implementation performs run-time checks, just to be sure, it will raise a "type fault" exception, contradicting the well-typedness theorem. In not, it might mis-interpret the bit string.

The problem was caused by the copying of non-values. ((*ref* ...) is a computation that has not yet resulted in a value.) As a result, the type-checker checks as if there were *several* values, which causes it to go awry. One solution to this problem is to require that the expression bound by **let** be a *syntactic* value. This way, type-sharing and execution sharing are never conflated. Andrew Wright proved that this simple solution is correct and practical [10].

Incidentally, we notice this problem immediately in an explicitly polymorphic system, since we would have to write

(**let** ((*f* (Λ α ...)))
  ...)

The Λ-notation immediately clarifies that the right-hand side of the **let** is a value, yet that it is not the intended reference value. The evaluation would create two reference cells, which avoids the type-fault, but is not what the programmer wanted.

17.2 Implications of Types for Execution

In languages like TLC and C, types are essentially a description of the number of bits needed for the storage of an object. During the past few chapters, we have made three significant additions to the type system that give it significant abstraction capabilities not found in TLC.

1. Consider a **datatype** declaration like

```
(datatype CI
  ((cons1 (int ...))
   (cons2 (char ...))))
```

Say we have a procedure (**lambda** (*x CI*) . . .). How much space do we need to allocate for *x*? Since, in general, we cannot tell which variants will be passed in, we will need to allocate space for the largest of them of them. (This is akin to *union*'s in Pascal.) One common technique is to use an indirection so that *x* occupies the same amount of space no matter which variant is passed in: one size then fits all. This is commonly known as *boxing*.

2. Implicit polymorphism similarly forces boxing.

3. Finally, consider a procedure such as an explicitly polymorphic identity function:

```
(define I
  (Λ (α)
    (lambda (x α)
      x)))
```

To use *I* on 5, we need to instantiate it first, as in ([*I int*] 5). We can then create a version of *I* that accepts integer-sized inputs which can be used with integers; and likewise for other types. Thus, explicit polymorphism is a way of making the execution time of implicit polymorphism tolerable, by saving (some of) the expense of boxing.

Exercise: Why do we not have the same problem in C++?

17.3 The Two Final Pictures

We have come to the end of our exploration of types, so it would be worthwhile to briefly summarize what we have examined. We shall summarize along two lines: the power and the landscape of typed languages.

17.3.1 Power

Is TLC a good programming language? Surely not — we can't even express recursion in it. One way out is to add *reclet*. Even explicit and implicit polymorphism, while extending the type system, can't express recursion. On the other hand, **datatype** *can* be used to program recursion. Thus, the design of typed languages is a very subtle problem, with some decisions having significant and overarching effects.

17.3.2 Landscape

The universe of typed programming languages is tri-polar. On one hand, we have languages like C, which have simple type systems and are unsafe. On the other hand, languages like SML and Haskell are polymorphic and safe. An extreme strain of this latter class is ML 2000, which is being designed atop an explicitly polymorphic system. (Languages like C++, Modula-3 and Java sit betwixt these poles, varying both in the power of their type systems and in their safety.)

Finally, on the *other* hand, we have Scheme, which is uni-typed (in the sense of ML), and is also safe. For languages like Scheme, a "type" system like set-based analysis appears to be more appropriate both at supporting the programmer's intuition and for program optimization. The advantage of Scheme's value-set system over ML's rigorous type system is that all values live in the same type space. Hence, it is possible to circumvent the *implied* type system if a programmer thinks that doing so is correct. In C this requires casting and is available because C is unsafe. In ML, this requires copying and/or injection/projection over datatypes. In sum, Scheme is safe, yet it allows all the tricks C programmers customarily use to evade the straight-jacket of the type system.

In summary, we have the following landscape:

C
simply typed
unsafe

Scheme
uni-typed/datatyped
safe

C++

Modula-3

Java

ML
implicitly polymorphic
safe

ML 2000
explicitly polymorphic
safe

With this, we conclude our survey of types.

18 The Meaning of Function Calls

There are at least three motivations for examining in depth what function calls mean.

1. In LC, a function call looks like (*f a*), and we interpret it as

```
((Eval f env) (Eval a env))
```

   which relies on two things: that functions in program text are represented as procedures in Scheme, and that application in the source is performed through a function call in Scheme. As an alternative, we could choose to represent functions as a data structure, and write

```
(Apply-closure (Eval f env) (Eval a env))
```

   Though this appears to abstract over all dependencies on Scheme, it does not: for instance, we still rely on Scheme's function call mechanism for much of the interpreter (such as the calls to *Apply-closure* or *Eval*). Indeed, this reliance pervades the interpreter. Since there is no direct analog to Scheme's procedure call mechanism in most machine instruction sets, we must find a better way to explain function calls if we want a primitive explanation of our features.

2. What does (*error* …) mean? So far, we have encoded *error* using **letcc**, but we would prefer a more direct explanation.

3. Finally, what does **letcc** itself mean? That too has been modeled with **letcc**, which is not satisfying.

Consider the following procedure, which computes the product of a list of numbers:

```
(define Pi
  (lambda (l)
    (cond
      ((null? l) 1)
      (else (* (car l) (Pi (cdr l)))))))
```

Now suppose the argument *l* may be corrupt and may contain non-numbers. We wish to change *Pi* so that, if the list contains only numbers, it returns their product; otherwise, it returns the first non-number that it encounters in the list. We could use an accumulator to do this:

```
(define Pi-2
  (lambda (l)
    (letrec ((Pi/acc (lambda (l acc)
                       (cond
                         ((null? l) acc)
                         ((number? (car l))
                           (Pi/acc (cdr l) (* (car l) acc)))
                         (else (car l))))))
      (Pi/acc l 1))))
```

Suppose *Pi-2* is passed a corrupt list. In that case, the helper function would have multiplied all the numbers found until the erroneous input is encountered. To avoid the wasted multiplications, we can defer the multiplication until we are certain the list has been completely traversed and found to be a legal input. The multiplication is delayed by wrapping it in a thunk:

```
(define Pi-3
  (lambda (l)
    (letrec ((Pi/acc (lambda (l acc)
                       (cond
                         ((null? l) (acc))
                         ((number? (car l))
                          (Pi/acc (cdr l)
                            (lambda () (* (car l) (acc)))))
                         (else (car l))))))
      (Pi/acc l (lambda () 1)))))
```

This program does indeed avoid unnecessary multiplications. However, suppose we were to modify the $*$ primitive so that it *prints* out its arguments before returning their product; then the three programs would not all produce the same (printed) output on legal lists.

Exercise: What will the outputs be? Will any two be the same? Use the reduction rules to determine what they will print.

The upshot is that an *intensional* aspect of the program's behavior has not been preserved. To get back the same order of evaluation while still deferring computation, we use a transformation called *continuation-passing style* (CPS), wherein we replace the thunk with a "promise" that the rest of the computation has been completed before it is evaluated. For example:

```
(define Pi-4
  (lambda (l)
    (letrec ((Pi/k
               (lambda (l k)
                 (cond
                   ((null? l) (k 1))
                   ((number? (car l))
                    (Pi/k (cdr l) (lambda (rp) (k (* (car l) rp)))))
                   (else (car l))))))
      (Pi/k l (lambda (x) x)))))
```

Why does the (**else** ...) clause work? It is because *Pi-4* is *tail-recursive*; hence, the value returned by the **else** clause is guaranteed to return directly to the caller of *Pi-4*.

It is worthwhile to note that, in this case, we have converted a properly recursive procedure into a tail-recursive one. If we can do this for *all* procedures, we will have succeeded at having explained recursion in terms of tail-recursion (and closure-creation, *etc*). Indeed, any compiler for a language that allows proper recursion must do something similar; thus, studying CPS can provide us with insight into and techniques for designing compilers.

The following steps must be taken to CPS a program:

1. Add an extra parameter to every recursive function. Call this parameter *k*. (The letter *k* is traditionally used to recognize the Teutonic contributions to this area.)

2. Each clause in a conditional is treated separately:
   (a) For each result *a*, write (*k a*).
   (b) For each non-recursive case, pick one recursive call. Make the result for the clause that of a recursive call which takes an extra argument, which is of the form (**lambda** (*rorec*) ...) ("*rorec*" is an abbreviation for "result of the rest of the computation"). The rest of the original contents of that clause are placed inside the (**lambda** ...), with the recursive call being replaced by a call to *k*.

We illustrate this with an example:

```
(define Pi*
  (lambda (t)
    (cond
      ((leaf? t) t)
      (else (* (Pi* (left t))
               (Pi* (right t)))))))
```

Then the CPS'ed version, *Pi*/k*, is

```
(define Pi*/k
  (lambda (t k)                    ; rule 1
    (cond
      ((leaf? t) (k t))            ; rule 2a
      (else                        ; rule 2b
        (Pi*/k (right t)
          (lambda (rorec)
            (k (* (Pi* (left t)) rorec))))))))
```

However, there is still a call to *Pi** left, which needs to be converted. The entire body of the continuation is converted to

```
(Pi*/k (left t)
  (lambda (x)
    (k (* x rorec))))
```

so that the final output is

```
(define Pi*/k
  (lambda (t k)                    ; rule 1
    (cond
      ((leaf? t) (k t))            ; rule 2a
      (else                        ; rule 2b
        (Pi*/k (right t)
          (lambda (rorec)
            (Pi*/k (left t)
              (lambda (x)
                (k (* x rorec)))))))))))
```

NOTE: In the process of CPS-ing *Pi**, we have made an explicit decision about the order of evaluation: specifically, that the right child will be traversed before the left.

19 Explaining Continuations and Errors

Our goal is to produce a tail-recursive interpreter, which we will use to understand the meaning of *error* and **letcc**.

To re-cap the previous section, during its evaluation, every phrase will be surrounded by some computation that is waiting to be performed (and, typically, that depends on the value of this phrase). This remaining computation is known as an *evaluation context*. Turning this evaluation context into a function is the act of making the continuation explicit.

For instance, in

```
(+ (* 12 3) (- 2 23))
```

the evaluation context of the first sub-expression (assuming it is evaluated first) is

```
(+ ◇ (- 2 23))
```

(where we pronounce ◇ as "hole"), so the reified version of this context is

```
(lambda (x) (+ x (- 2 23)))
```

Exercise: Can (**lambda** (*x*) ... ◇ ...) be a valid evaluation context?

19.1 Modeling Errors

Let us consider an interpreter for LC:

```
(define Eval
  (lambda (M env)
    (cond
      ((var? M)
        (lookup M env))
      ((lam? M)
        (make-closure M env))
      ((app? M)
        (Apply (Eval (app-rator M) env) (Eval (app-rand M) env)))
      ((add? M)
        ...)
      ...)))
```

In this interpreter, we see both the creation of new continuations and the use of the (implicit) continuation. New continuations are created in the code for applications, at the evaluation of the sub-expressions and again in the call to *Apply*. The other two clauses shown use the current continuation by passing it a value.

We now use the standard technique for transforming Scheme code to transform the interpreter into CPS:

```
(define Eval/k
  (lambda (M env k)
```

```
    (cond
      ((var? M) (k (lookup M env)))
      ((lam? M) (k (make-closure M env)))
      ((app? M) (Eval/k (app-rator M) env
                  (lambda (rator-v)
                    (Eval/k (app-rand M) env
                      (lambda (rand-v)
                        (Apply/k rator-v rand-v k))))))
      ...)))
```

We need to similarly transform *Apply*. Whereas before we had

```
(define Apply
  (lambda (f a)
    (cond
      ((closure? f)
        (Eval (body-of f)
          (extend (env-of f) (param-of f) a)))
      (else ...))))
```

we now have

```
(define Apply/k
  (lambda (f a k)
    (cond
      ((closure? f)
        (Eval/k (body-of f)
          (extend (env-of f) (param-of f) a)
          k))
      (else ...))))
```

NOTE: None of the *Apply* procedures take an environment as an argument; instead, they choose to use the one stored in the closure. However, it is possible to imagine a different semantics that passes on the current environment to *Apply*, which then passes it on for the remainder of the evaluation. Such a system is said to have *dynamic binding*, as opposed to the *static binding* used here.

Exercise: Implement dynamic binding.

We have intentionally left the fall-through case of the **cond** expressions in the *Apply* procedures empty. However, we know from before that there should be a call to *error* in that slot. Since we want to explain *error*, we need to determine how to signal an erroneous application.

Since the *k* argument always represents the entire evaluation context, there is no additional computation awaiting the value returned by the interpreter. Therefore, to ignore the pending computations and return a value directly, all that *error* needs to do in this interpreter is to return a value. This value is then returned to the user. (Note that this crucially depends on the fact that the interpreter is fully tail-recursive.) Therefore, the code for *Apply/k* could be written as

```
(define Apply/k
```

```
(lambda (f a k)
  (cond
    ((closure? f)
      (Eval/k (body-of f)
        (extend (env-of f) (param-of f) a)
        k))
    (else 'ouch!))))
```

Note that the **else** clause above is "returning" at the meta-level, not at the level of LC. *A function return in LC is modeled by passing the result of evaluating the function call to the appropriate continuation.*

We have teased out control entirely and made it a distinct entity in the interpreter. Thus, we can use it as before, we can ignore it, or we can harness it in new ways, as we will see below.

Exercise: What value would we use as the initial continuation?

19.2 Modeling Continuations

In our earlier interpreters, we modeled **letcc** by appealing to the **letcc** form at the meta-level:

```
(cond
  ...
  ((letcc? M)
   (letcc k (Eval (body-of M)
                  (extend env (label-of M) k))))
  ...)
```

Since we want **letcc** to bind a program variable to the rest of the computation, we could instead write, in *Eval/k*,

```
(cond
  ...
  ((letcc? M)
   (Eval/k (body-of M)
     (extend env
       (label-of M) k)
       k))
  ...)
```

This rewrite makes manifest an interesting feature of control. Just as stores were duplicated when we made them explicit, here we have *two* uses of *k*. In the case of stores, we hypothesized that we could have an operator that "forgot" the side-effects performed in the evaluation of an expression, but we found no use for such an operator. Now, however, we can save the evaluation context, and return to it — ignoring any intermediate context — if we wish. Furthermore, since this evaluation context is bound in the environment, we can return to it whenever we choose to, even *outside* the lexical context of the **letcc** expression. One example of where this might be useful is in the implementation of *context switches* during multitasking, where we periodically store the current context and return to it later.

Hence, whereas before control was single-threaded, and could be implemented as a stack, it is now tree-shaped and cannot be implemented in that manner.

19.3 Eliminating Closures

Earlier, we decided to eliminate meta-language closures from our interpreter. Now, we have reintroduced them in the process of CPSing our interpreter, *ie*, in the action for applications. As before, we shall model this process of closure creation abstractly by using the procedure *Push*. Similarly, we will abstract over the use of the continuation — to "return" values — as the procedure *Pop*.

```
(define Eval*/k
  (lambda (M env k)
    (cond
      ((var? M) (Pop k (lookup M env)))
      ((lam? M) (Pop ...))
      ((app? M)
        (Eval (app-rator M) env
          (Push 1 M env k)))
      ...)))
```

To begin with, these abstractions can map to the current implementation method:

```
(define Push
  (lambda (name M env k)
    (lambda (x)
      (Eval/k (app-rand M) env
        (lambda (y)
          (Apply/k x y k))))))

(define Pop
  (lambda (k v)
    (k v)))
```

Note, further, that *Push* has a call to *Eval/k*; we rewrite this as

```
(Eval/k (app-rand M) env (Push 2 x k))
```

In all, there is a finite number of locations that create continuations in the interpreter, so we can write a procedure that creates each one of these. Thus, *Push* can be rewritten as

```
(define Push
  (lambda (name . pv)
    (cond
      ((= name 1)
        (let ((M ...pv...) (env ...pv...) (k ...pv...))
          (lambda (x) (Eval/k (app-rand M) env
                        (Push 2 x k)))))
      ((= name 2)
        (let ((x ...pv...) (k ...pv...))
          (lambda (y) (Apply/k x y k)))))))
```

Finally, the remaining reliance on **lambda** can be removed by replacing it with a call to *list* instead. In the process, *Pop* has to become more elaborate: in particular, it needs to dispatch on the "name" of the current continuation. This is analogous to the change made in *Apply* when we moved from a meta-level to a structure-based representation of closures in our original interpreters.

20 The True Meaning of Function Calls

In this section, we will make repeated use of three key transformations used in building software: representation independence, switching representations, and CPS.

We begin by CPSing our interpreter:

```
((var? M) (k (lookup M env)))
((app? M) (Eval (app-rator M) env
            (lambda (f)
              (Eval (app-rand M) env
                (lambda (a)
                  (Apply f a k))))))
```

We then make it representation independent,

```
((var? M) (Pop k (lookup M env)))
((app? M) (Eval (app-rator M) env
            (Push-app-f M env k)))
```

where

```
(define Push-app-f
  (lambda (M env k)
    (lambda (f)
      (Eval (app-rand M) env
        (lambda (a)
          (Apply f a k))))))
```

and

```
(define Pop
  (lambda (k v)
    (k v)))
```

Then in *Push-app-f* we rewrite the last procedure to

```
(lambda (a)
  (Push-app-a f k))
```

with

```
(define Push-app-a
  (lambda (f k)
    (lambda (a)
      (Apply f a k))))
```

Now in *Push-app-f*, which now looks like this,

```
(define Push-app-f
  (lambda (M env k)
    (lambda (f)
      (Eval (app-rand M) env
        (lambda (a)
          (Push-app-a f k))))))
```

we replace the **lambda** with (*list* 'app-f *M env k*) to get

```
(define Push-app-f
  (lambda (M env k)
    (list 'app-f M env k)))
```

and *Push-app-a* is modified similarly, so we assume that all continuations have distinguishing tags.

Exercise: The last time we wanted to make our representations independent of the underlying Scheme representations, we created *make-closure* and expressed closures in terms of that. Since our continuations are quite similar to closures, why can't we use *make-closure* to represent continuations as well?

Given these definitions, we can now write down the code for *Pop*. Recall that *Pop* needs to examine the tag at the front of the continuation to determine what to do.

```
(define Pop
  (lambda (k v)
    (if (list? k)
      (case (car k)
        ((app-f) (let ((M (cadr k))
                       (env (caddr k))
                       (k (cadddr k)))
                   ((lambda (f)
                      (Eval (app-rand M) env
                        (Push-app-a f k)))
                    v))))
      (k v))))
```

(Note that we should be able to transform the continuations one at a time. Hence, *Pop* includes both methods of invoking continuations.) But we know that ((**lambda** ...) ...) is just **let**, so we can rewrite the *app-f* case as:

```
(let ((M (cadr k))
      (env (caddr k))
      (k (cadddr k)))
  (let ((f v))
    (Eval (app-rand M) env
      (Push-app-a f k))))
```

We will find it convenient to also add a continuation with tag *stop* which is used to halt computation. This is the default initial continuation.

20.1 From Tail-Recursion to Register Machines

A register machine has a number of registers, an unbounded amount of memory, assignment to memory, assignment to registers, and a goto statement.

By now, the interpreter has only tail recursive calls to *Eval* and to *Pop*; the different variations on *Push* are all trivial. We now undertake the following steps:

1. Making the parameters of TR functions into global variables, such as

```
(define =M=)
(define =env=)
(define =k=)
```

   Thus, we can eliminate all parameters for procedures by assigning to the parameter registers instead. For instance, the interpreter looks like

```
(define Eval
  (lambda ()
    (cond
     ...)))
```

2. "Initializing" the parameters (registers). We place the term in quotes since it needs to be done each time the procedures are invoked (see below). In addition, we add a procedure to initiate the interpreter:

```
(define Main
  (lambda (M)
    (set! =M= M)
    (set! =env= (empty-env))
    (set! =k= 'stop)))
```

3. Calling functions without arguments after assigning the arguments into the appropriate registers:

```
(define Eval
  (lambda ()
    ((var? =M=)
      (set! =k= =k=)                    ; redundant
      (set! val (lookup =M= =env=))
      (Pop))
    ((app? =M=)
      (set! =k= (Push-app-f =M= =env= =k=))
      (set! =M= (app-rator =M=))        ; destroys M!
      (set! =env= =env=)
      (Eval))
    ...))

(define val)

(define Pop
  (lambda ()
    (case (car k)
      ((push-app-f) (let ((M ...)
                          (env ...)
                          (k ...)
```

```
                (f val))
              (set! =k= (Push-app-a f k))
              (set! =M= (app-rand M))
              (set! =env= env)
              (Eval)))
  ...)))
```

Now we're left only tail-recursive function calls. We can test this in the following manner. First, we define *Goto*:

```
((call/cc
  (lambda (k)
    (set! Goto k)
    (lambda () "hello world"))))
```

Now for each tail call, instead of writing (*proc*), we write (*Goto proc*) instead.

Exercise: Why is *Goto* an appropriate name for the above continuation? What does the (*proc*) to (*Goto proc*) transformation help prove, and how?

At this point, we are left with only the following: **cond**, **set!**, selector functions, *Goto* and the *Push-app-∗* procedures. All of these can be trivially implemented at the machine level: most correspond directly to machine instructions, while the *Push* procedures allocate enough space to put the pointers in, and return a pointer to that newly allocated memory. We have already seen how to implement such procedures before with an array of memory. Hence, the only unnatural assumption left in our interpreter is that we have an unlimited amount of memory.

Example

Consider the factorial function:

```
(define !
  (lambda (n)
    (if (= n 0)
      1
      (* n (! (- n 1))))))
```

First, we CPS it:

```
(define !
  (lambda (n)
    (!/k n (lambda (x) x))))

(define !/k
  (lambda (n k)
    (if (= n 0)
      (k 1)
      (!/k (- n 1) (lambda (v) (k (* n v)))))))
```

and then make it representationally independent:

```
(define !
  (lambda (n)
    (!/k n Stop)))

(define !/k
  (lambda (n k)
    (if (= n 0)
      (Pop k 1)
      (!/k (− n 1) (Push n k)))))
```

where

```
(define Stop
  (lambda (x) x))

(define Push
  (lambda (n k)
    (lambda (m) (k (∗ n m)))))

(define Pop
  (lambda (k m)
    (k m)))
```

We can represent our continuations more directly by using a list. This change only involves modifying the helper functions, leaving the main code untouched:

```
(define Stop
  '())

(define Push
  cons)

(define Pop
  (lambda (k m)
    (if (null? k)
      m
      (Pop (cdr k) (∗ (car k) m)))))
```

It is easy to see that the lists we get will always consist of numbers. In other words, *Pop* is *Pi* in accumulator style. Hence, we can make the representation even more concise:

```
(define Stop
  1)

(define Push
  ∗)

(define Pop
  ∗)
```

and voilà, we get the C version of *!* in a loop.

21 How To Eat Your Memory and Have It, Too

From the previous section, we can see that our machine requires five registers:

1. =*M*=: the program text
2. =*env*=: the lexical context [variable-value pairs]
3. =*k*=: the control context [list of frames]
4. =*val*=: the result value from evaluating the contents of =*M*=
5. =*param-val*=: the value of a function's parameter

=*M*= is a pointer into the program code. =*k*= holds a stack, which can be implemented as a pointer into a separate array. The other three are registers that (may) directly point to allocate data structures such as closures and lists.

Let us name the following expressions

M1 = (**lambda** (*x*) (+ *x* 3))
M2 = (**lambda** (*f*) (+ (*f* 7) 4))
M3 = (**lambda** (*z*) (− *z* *y*))

and consider the evaluation of

(*M1* (*M2* (**let** (*y* 2) *M3*)))

We will study the evaluation of this expression by looking at "snapshots" of the machine at various stages.

Snapshot 1 We have evaluated *M1* and are in the process of evaluating the argument to the resulting closure.

=*k*= = [*appR* $\longrightarrow$ <*M1*,*empty*>]
=*env*= = *empty*
=*val*= = <*M1*,*empty*>

where =*val*= shares its contents with =*k*=.

Snapshot 2 We have evaluated the left and right terms from Snapshot 1, and are about to apply the closure formed from *M2*.

=*k*= = [*appR* $\longrightarrow$ <*M2*,*empty*> , *appR* $\longrightarrow$ <*M1*,*empty*>]
=*env*= = *empty*
=*val*= = <*M3*,[<*y*,2>]>

Snapshot 3 We are just done evaluating the subtraction inside *M3*, which is bound to *f*.

=*k*= = [+*R* $\longrightarrow$ 4 , *appR* $\longrightarrow$ <*M1*,*empty*>]
=*env*= = [<*z*,7> , <*y*,2>]
=*val*= = 5

Note that we have opened up the environment of the closure bound to *f* in showing the value of =*env*=.

Snapshot 4 We are in the midst of the addition inside the closure <*M1*,*empty*>; the *x* has just been evaluated.

=*k*= = [+*R* $\longrightarrow$ 3]

=env= = [<*x*,*9*>]
=val= = 9

However, recall that there are several old fragments of environment still to be found in memory, such as [<*z*,*7*> , <*y*,*2*>] from Snapshot 3.

If we look carefully in the final step, there are many items that were formerly in the environment that are unnecessary for the remaining evaluation. However, these unnecessary items are still present in memory and could potentially cause our program to exhaust available memory before finishing its task. Hence, we should try to recycle such unnecessary memory. *Id est*:

1. Memory is a forest, rooted in registers.
2. As the computation progresses, some portions of it become unreachable.
3. Therefore, memory is reusable.

Assume we divide up available memory into two halves, called "memory 1" and "memory 2". Say we begin by allocating in memory 1, and hit the boundary. Then we can switch our current half to memory 2, copy the tree of reachable memory from the memory 1 into memory 2, and proceed with the computation. This copying is done by picking a register, each one in turn, and walking pointers into memory until we hit a *cons* cell; we copy this into the new memory 2, and repeat the procedure along each component of the cell. The process is repeated when memory 2 is exhausted, switching the rôles of the two parts.

This method might make intuitive sense, but what if we have sharing in our language? In LC, we currently have no way of checking sharing constraints (as with *eq?* in Scheme), but it is reasonable to assume we might be called upon to do so. In addition, if we duplicated objects, we would in fact use more space in the new half than in the old one, which would ruin the purpose of our attempt at recycling memory. To prevent this, when we visit a cell, we have to indicate that it has been *forwarded*; then, if it is visited again, the appropriate sharing relationship can be mimicked in the new half.

Thus, with the help of this process, which is called *garbage collection*, if the two memory banks are of equal size, and if there are indeed unreachable objects in the exhausted space, then we will have space left over in the new bank, and we can proceed with our allocation. However, there are two problems:

1. What if everything is reachable? Then we are forced to signal an error and halt computation. (Note that this doesn't mean there aren't "unusable" objects in memory, just that our notion of reachability isn't strong enough to distinguish these objects. The objects that are truly necessary are said to be *live*.)
2. The collector itself needs space for recursion and computations. We know we can get rid of recursion using CPS, which also tells us how many registers we need (which is fixed). The remaining variable is the depth of the stack, but this is proportional to the depth of the tree being copied. Using these insights, it is possible to write the collector that uses a small, fixed amount of additional memory.

A simple model of the garbage collector might look like this:

```
(define gc
  (lambda (ptr)
```

```
(cond
  ((null? ptr) null)
  ((cons? ptr) (cons_mem1 (gc (car_mem2 ptr))
                          (gc (cdr_mem2 ptr))))))
```

but this loses sharing. So we have to break *cons_mem1* up into its two constituent parts: allocation and initialization.

```
((cons? ptr)
 (let ((new (alloc ...)))
   (make-as-forwarded ptr)
   (init_mem1 new (gc (car_mem2 ptr)) (gc (cdr_mem2 ptr)))
   new))
```

However, this still doesn't check for forwarding. A simple modification takes care of that:

```
((cons? ptr)
 (if (forwarded? ptr)
   ...
   (let ((new (alloc ...)))
     (make-the-orange-thing)
     (init_mem1 new (gc (car_mem2 ptr)) (gc (cdr_mem2 ptr)))
     new)))
```

21.1 Perspective

In summary, the traditional view of garbage collection is roughly as follows:

1. Reachability corresponds to liveness.
2. Non-reachable memory can be garbage collected.
3. The registers are the roots from which to perform the sweep.

Recently, a new view of garbage collection has been emerging. In this view,

1. Every program evaluation state in the register machine corresponds to memory, registers and the program text.
2. It is impossible to decide whether any given cell in some machine state is live or dead.
3. Every algorithm that conservatively identifies live cells (*ie*, does not mistakenly claim some cell to be dead when it is useful) is a garbage collection algorithm.

The new view of gc has given raise to new gc algorithms. The new algorithms reconstruct the types of all phrases, including memory cells, at run-time and use type information to determine which cells are live or dead. For example, if an implicitly polymorphic system is used and a cell has type α, the program evaluation will work for all possible values in that cell. In particular, it will work if the cell's content is replaced by the null pointer. Doing so frees all memory that the cell (may) point to.

The new view is *logical*: it distinguishes between truly live and provably live cells, between truth and provability. As always, the latter is an approximation of the former. It is another indication of how tightly logic and computation are intertwined.

22 Adequacy, Compilers, Optimizers and Observational Equivalence

22.1 Adequacy

During the course of the semester, we have developed a variety of evaluators for (T)LC. Some of these are *eval-subst*, *eval-env*, *eval-cps* and *eval-store*. All of these purport to implement the language in the same manner, but what does this even mean? Since they all are written in the mathematical subset of Scheme, it should be possible to use mathematics to answer this question.

Mathematically, an evaluator for a language L is a function from programs to answers:

$$\textit{Eval-L} : \textit{Program-L} \longrightarrow \textit{Answer}$$

Now we can ask, when are these functions equal?

Attempt 1 For all programs P, we expect

$$\textit{Eval-subst}\ (P) = \textit{Eval-env}\ (P)$$

But say we pick P to be ((**lambda** (x) (x x)) (**lambda** (x) (x x))). Since the evaluators do not even terminate on this program, it makes no sense to compare the (non-existent) answers.

Hence, we clearly cannot use mathematical equality, $=$. We could instead use Kleene-equality, $\cong$, which requires that if one side is *defined* (or *exists*) then so is (does) the other, and that they then both be equal.

Attempt 2 Our first attempt failed because the evaluators were assumed to be total functions over their domains, which they are not. But a function is merely a set of elements – a relation – with constraints. Hence, to mirror the use of $\cong$, we could instead require

$$\textit{Eval-cps} = \textit{Eval-env}$$

whereby we require that the *sets* be equal. So we have fixed the earlier problem by not requiring the domain to be the set of all programs.

Now consider the program (**lambda** (x) x). What does this evaluate to in each evaluator?

(*Eval-subst* '(lambda (x) x)) = '(lambda (x) x)
(*Eval-env* '(lambda (x) x)) = (**lambda** (d) (*Eval* 'x (*extend mt* 'x d)))
= (**lambda** (d) d)
(*Eval-cps* '(lambda (x) x)) = (**lambda** (d) (k d))

These are not the same elements! This shows that we must also carefully define the set of answers, which is the range of the evaluator.

Attempt 3 We can fix the ranges by providing a wrapper for each evaluator such that, if the original evaluator reduces the program to a number, the wrapper returns that number; if the program reduces to any closure, the wrapper returns the special token *closure*. Only after doing this does the equation

$$\textit{Eval-cps} = \textit{Eval-env}$$

even make sense.

Note that we have thus far said nothing about how we would *prove* such a result; we have only clarified our definition until it makes sense. In practice, such a proof would be done over the structure of terms (and is quite complicated). This property is called *adequacy*.

The word "adequacy" reflects the following viewpoint: every language consists of a syntax and an evaluator that maps programs to answers, which requires the specification of what programs are, what answers are, and how to go between the two. For example, the syntax of the language might be that of TLC, while the evaluator is *Eval-subst*. Then, we want to be able to say that any other evaluator "adequately expresses" the behavior of *Eval-subst*, *ie*, that we get the same answer (if any) from either evaluator.

22.2 Compilation

A *compiler* is a function from programs to programs. Whereas an evaluator takes a program in language L and produces an answer (if any), a compiler takes a program in L and produces a program in some language M. This distinction is easily seen from the notation itself:

$$compiler_{L,M} : Program_L \longrightarrow Program_M$$

$$evaluator_L : Program_L \longrightarrow Answer$$

There is, however, a relationship between compilers and evaluators. Since every well-defined language is equipped with some evaluator (semantics), we know that M has an evaluator and that the compiler can only be correct if it satisfies the following equation:

$$evaluator_M \circ compiler_{L,M} = evaluator_L$$

(where $\circ$ represents the composition operator).

The question naturally arises of how we can obtain a compiler for a pair of languages (*par abus de langage*, the phrase "a compiler for language L" is sometimes used; in these cases, M is implicitly understood). One approach is to consider a program that we believe to be a compiler, and set out to prove the above equation, which is a special form of adequacy; but for any realistic pair of languages, it is virtually impossible to construct the purported compiler without any thought to the proof, and then set about trying to prove it correct.

Hence, effort has been invested in *deriving* compilers, with the above equation being the guiding principle, where correctness of the compiler is maintained at each stage. For instance, (**lambda** (*x*) *x*) is a correct compiler (for $L = M$). Then the code of $evaluator_M$ is carefully considered, and work that can be done now is shifted into the compiler. For instance, say the evaluator is *eval-env*. If we examine the environment, we see that one part of it (the lexical structure) can be deduced from a program's text, while the values only become known at run-time. Hence, the environment can be split into two parts, the evaluator can be curried, and calls of the form (*Eval M env*) can be rewritten as ((*Eval M senv*) *denv*). Code relying entirely on static components can then be moved into the compiler.

Slogan: *The proof* is *the construction.*

22.3 Optimization and Observational Equivalence

Suppose a program used the bubble-sort routine. A compiler could recognize this usage and replace it with an appropriate combination of quick-sort and insertion-sort, say. If we did not examine *intensional* properties such as the duration of execution or the the number of hits in the cache, we would observe no difference in the execution of the program.

It is helpful to formalize the notion that, no matter what context we are in, we cannot distinguish between two expressions. We will do it as follows.

First, we define *contexts*, which are expressions with a "hole" (written as []) for some subexpression. The grammar for contexts is

$$
\begin{aligned}
C ::= &\ [] \\
&\mid (\textbf{lambda}\ (x)\ C) \\
&\mid (C\ M) \\
&\mid (M\ C) \\
&\mid (+\ C\ M) \\
&\mid (+\ M\ C)
\end{aligned}
$$

where M is an arbitrary LC expression. Then the hole is "filled" by substituting an expression in its place; to substitute M into C's hole, we write $C[M]$. Note that contexts are not the same as evaluation contexts, since we can have a hole within a (**lambda** ...) body.

We then define that M and N are *observationally equivalent* if and only if

$$Eval\ (C[M]) \cong Eval\ (C[N])$$

for *all* contexts C.

Note that observational equivalence is really a relation between program phrases (which could be open). It is an equivalence relation; technically, it is called a *congruence* relation, since it "commutes" with syntax. *Every well-defined language is equipped with an equational theory*. Based on this theory we can *always* replace equals with equals, though the concept of equality radically changes as the language changes. An ideal compiler replaces a phrase with an observationally equivalent phrase that has better *intensional* properties.

However, observational equivalence is a very strong relationship. It requires a complete equality of behavior of two terms, including their termination behavior. Yet, aggressive compilers tend to ignore the termination behavior of phrases. That is, replacements are sometimes made such that

$$Eval\ (C[M]) = a \Longrightarrow Eval\ (C[N]) = a$$

where the implication is uni-directional. We no longer have symmetry, so the relationship is no longer an equivalence but rather is a pre-order ($\preceq$).

In practice, in addition to ignoring the non-terminating behavior of some phrases, a compiler may also ignore the "exception behavior" of a phrase. That is, it may replace a phrase M with a phrase N that raises different exceptions from those in M or none at all in certain situations. As a result, there is no simple mathematical description of the activities of aggressive optimizers; on the practical side, this lack of a concise description makes it impossible to debug aggressively optimized programs.

23 Control Versus State: CPS and SPS

In the direct interpreter, *Eval* takes *M* and *env* and eventually produces a value (if *M* reduces to one). When we add continuations to the language, we add an entry in the interpreter such as

```
((letcc? M) (call/cc (lambda (k)
                        (Eval (letcc-body M)
                          (extend env (letcc-label M) k)))))
```

This gives us a meta-explanation of **letcc**, but is not particularly satisfactory for understanding what **letcc** really does.

In the CPS'ed interpreter, *Eval* takes *M*, *env* and *k*. Eventually, the computation reduces to (*k value*), and *value* is the result of the expression. In this world, we have

```
((lam? M) (k (make-closure M env)))
((app? M) (Eval (app-rator M) env
            (lambda (f)
              (Eval (app-rand M) env
                (lambda (a)
                  (f a k))))))
```

In this setting, the evaluation of **letcc** can be explained without resorting to the eponymous operator in the meta-language:

```
((letcc? M) (Eval (letcc-body M)
              (extend env (letcc-label M)
                (lambda (a b)
                  (k a)))))
```

The representation of the continuation as a meta-closure (or "record") immediately clarifies two important aspects of continuations:

1. the representation of continuations is (must be) compatible with the representation of closures; and,

2. just like a closure, a continuation accepts the current control context as an argument (*b*), but it uses the old continuation (*k*) instead.

Exercise: Can we write (*b* (*k a*)) in the code for **letcc**?

Recall that an interpreter must eventually return answers, which may be distinct from values. The expression only makes sense if the two sets are equal, *ie*, if the result of applying the functional representation of the rest of the computation to a value is a value.

We have now clarified control, but we still do not have a clear idea of what happens to effects on the environment. Since *k* is a closure, it closes over *env*, but the above code doesn't explain what happens to effects. By converting the interpreter to store-passing style (SPS), we will be able to tease out this property, too.

An SPS evaluator takes *M*, *env*, a store *sto* and returns the final value and a new store.

```
((var? M) (pair (lookup_sto (lookup_env M env) sto) sto))
((app? M) (let* ((pair-1 (Eval (app-rator M) env sto))
                 (f (LEFT pair-1))
                 (sto (RIGHT pair-1))
                 (pair-2 (Eval (app-rand M) env sto))
                 (a (LEFT pair-2))
                 (sto (RIGHT pair-2)))
            (f a sto)))
```

Note that the initial store is passed to the evaluator to reduce the applicator; the new store obtained, not the original one, is passed recursively when reducing the applicand; and it is a new store again that is used when the function is finally applied. In this interpreter, we implement continuations as follows:

```
((letcc? M) (call/cc (lambda (k)
                       (Eval (letcc-body M)
                         (extend env (letcc-label M)
                           (new sto))
                         (extend sto (new sto) k)))))
```

While this clarifies the flow of stores, it still leaves unclear the relationship between continuations and stores, due to the explicit use of meta-continuations.

Hence, we consider the Continuation/Store-Passing Style ((CS)PS) interpreter. In this, the evaluator takes *M*, *env*, *sto* and *k*, and reduces to an application of (*k value sto*).

```
((lam? M) (k (make-closure M env) sto))
((app? M) (Eval (app-rator M) env sto
            (lambda (f sto-1)
              (Eval (app-rand M) env sto-1
                (lambda (a sto-2)
                  (f a sto-2 k))))))
((letcc? M) (Eval (letcc-body M)
              (extend env (letcc-label M) (new sto))
              (extend sto (new sto)
                (lambda (v sto dynamic-k)
                  (k v sto)))))
```

In the last call above, the store being used is the new dynamic store, while the continuation being applied is the old one captured in the closure. Hence we see that the flow of the store is actually different from the flow of the continuation in the presence of **letcc**.

23.1 Threads

A program consists of some control state and some collection of data. It is conceivable that there could be one collection of data, but several pieces of control operating on it. These multiple pieces of control are called *threads*. Each thread is free to perform different operations on the same collection of data.

We can easily extend our interpreters to handle a simple model of *multi-threading* (the process of programming with more than one thread). One might conceptually think of the multiple threads operating "in parallel", *ie*, all at the same time, but they cannot be implemented this way on a stock uniprocessor machine. Instead, we perform *time-slicing*: each thread is allowed to run for some duration of time after which it is paused; another thread takes control, and so on until all the threads are done.

To add threads to our interpreter, we need a timer mechanism to determine the end of time slices. A simple surrogate is to keep a counter that is initially set to some value (the *quantum* of time given to each thread each time around). Whenever the interpreter is entered, the counter is decremented. When it finally reaches 0, a designated procedure (the *timer interrupt handler*) is invoked. This handler is responsible for pausing the current thread, setting the quantum, and then resuming the next thread. More concretely, we might have a global timer variable

```
(define thread-timer 1000)    ; some initial value
```

and the evaluator would set and check it upon each invocation:

```
(define Eval
  (lambda (exp env)
    (set! thread-timer (− thread-timer 1))
    (if (≤ thread-timer 0)
      (thread-interrupt-handler))
    (cond
      ((var? exp) ...)
      ...)))
```

Keeping track of the threads, and processing them in order, is easy with a queue. But how do we pause and resume threads? This is easy to do with continuations: pausing corresponds to capturing the continuation of the thread, storing it in the queue and resuming some other thread (assuming it is represented as a continuation, which it is in this model); resumption consists of removing the continuation from the queue and invoking it with some (dummy) value.

To illustrate this, we present a simple timer interrupt handler which does all of the above. For humor value, it introduces some randomness into the duration of each quantum.

```
(define thread-interrupt-handler
  (lambda ()
    (call/cc
      (lambda (rest-of-thread)
        (enqueue! rest-of-thread thread-queue)
        (set! thread-timer (+ 1000 (random 1000)))
        ((dequeue! thread-queue) 'dummy)))))
```

Exercise: Why does returning a dummy value not cause any problems?

With this simple model, it is now easy to express a variety of paradigms. For instance, it is currently impossible for a thread to kill itself ("commit suicide") or to create a new thread ("fork"), but these operations are easily added.

Exercise: Add *fork* and *suicide*.

Threads lie at the heart of most modern operating system execution models. Our interpreter-based approach makes it easy to explore a host of such approaches.

It is important to note that threads introduce numerous difficulties if the language also includes mutation. Since the absolute order of execution of instructions in the program may no longer be determinate (*eg*, if there is a random component to the handler, as above), such errors may manifest themselves only in certain impropitious configurations, and may be extremely hard to reproduce.

References

[1] William Clinger and Jonathan Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[2] Matthias Felleisen. A lecture on the why of Y. Unpublished manuscript. `http://www.cs.rice.edu/~matthias/Papers/Y2.ps`.

[3] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, 1997.

[4] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[5] James Gosling, Bill Joy, and Guy Lewis Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[7] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[8] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, August 1986.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[10] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, 1995.