CS704: Lecture 10
Interpretation

Thomas Reps
[Based on notes taken by Suhail Shergill on February 10 and 12, 2010]

Abstract

This lecture discusses several kinds of interpreters for a simple functional programming language.

1 Introduction

In previous lectures about the λ-calculus, we considered an evaluation strategy based on the repeated use of a collection of rewriting rules. We now present some alternative—and more reasonable—execution mechanisms, which are based on using auxiliary data structures, in particular, an "environment" that holds bindings of names to expressions or values. Instead of evaluation by rewriting, the interpreters will perform a kind of "virtual rewriting" in which variables will be bound to certain values in a given context. Different variants of such an interpreter will exhibit different semantics, for example:

- call-by-value
- call-by-name
- call-by-need

For some programs, call-by-value returns $\perp$ (i.e., no answer), whereas call-by-name and call-by-need return an answer. The difference between call-by-name and call-by-need is that call-by-need will only evaluate certain expressions *once*, whereas call-by-name can end up reevaluating the same expression *multiple times*.

In later lectures, we will consider *partial evaluation*, which transforms programs of "type" $(T \times T) \rightarrow T$ to ones of type $T \rightarrow (T \rightarrow T)$. (Note that such a transformation makes sense because the set of functions $(T \times T) \rightarrow T$ is isomorphic to the set of functions $T \rightarrow (T \rightarrow T)$.) Note that the objective is not just to preserve the program's "type", but also to make sure that the transformation preserves the intended semantics of the original program (while allowing arguments to be provided one at a time in the transformed program).

The order in which we consider these various issues will be as follows: first, in this lecture, we discuss interpretation in the context of functional languages; second, we consider partial evaluation of a single-procedure imperative language; finally, we discuss partial evaluation for functional languages. (The techniques used for partially evaluating the [multiple] functions of a functional language provide insight into the similar techniques that are needed to handle multi-procedure imperative programs.)

2 A Word about Notation

Our subject language will be the Scheme0 language:

Data

1. Atoms:
    - Literals - A sequence of characters

- Numeric - Integers
- Booleans - T, NIL

2. S-expressions - These are binary trees with atoms for leaves. An S-expression is either an atom or a cons pairs (with a cons node denoted by a dot, e.g., (A . B)).
3. Lists - These are a subset of S-expressions (whereas S-expressions represent binary trees, lists are right-skewed trees). An expression of the form (A B) denotes the S-expression (A . (B . NIL)).

Operations for manipulating S-expressions

1. $cons : sexpr \times sexpr \rightarrow sexpr$
2. $car : sexpr \rightarrow sexpr$
3. $cdr : sexpr \rightarrow sexpr$
4. $atom : sexpr \rightarrow \{\text{T, NIL}\}$
5. $eq : sexpr \times sexpr \rightarrow \{\text{T, NIL}\}$

We will make a distinction between so-called “M-notation” (“Meta-notation”), in which atoms are written in lower-case and square brackets are used to denote the actual parameters of function calls (e.g., foo[e1,e2]), and “representation notation”, in which atoms are written in upper-case, expressions are written in prefix, and parentheses are used to surround expressions. For instance, the M-notation call foo[e1,e2] looks like (CALL FOO rep[e1] rep[e2])) in representation notation, where rep[e] converts expression e in M-notation into the corresponding representation-notation S-expression.

Note that all representation-notation objects are S-expressions.

In fact, we will overload rep[·] to work on programs and definitions as well as expressions:

	M	rep[M-object]
Prog →	Defn$_1$; … Defn$_k$	(rep[DEFN$_1$] … rep[DEFN$_k$])
Defn →	Name[Var$_1$, …, Var$_m$] ← Exp	(DEFINE (NAME VAR$_1$ … VAR$_m$) rep[Exp])
Exp →	n	n
|	an S-expression constant A	(QUOTE A)
|	var	VAR
|	**if** Exp$_1$ **then** Exp$_2$ **else** Exp$_3$	(IF rep[Exp$_1$] rep[Exp$_2$] rep[Exp$_3$])
|	Name[Exp$_1$, …, Exp$_k$]	(CALL NAME rep[Exp$_1$] … rep[Exp$_k$])
|	T	T
|	NIL	NIL
|	pattern-matching notation	—

As a shorthand, we allow the use of a Burstall-like pattern-matching notation in M-notation expressions. We will not define the corresponding representation-notation expressions, but rely on our intuition that expressions involving pattern-matching notation can be rewritten to ones that do not use pattern-matching notation. (For those interested in reading about how pattern-matching notation can be compiled to an efficient form for execution, see the papers by Wadler [2] and Pettersson [1].)

We can describe the semantics of the Scheme0 S-expression-manipulation primitives as follows:

$$
\begin{aligned}
cons[x, y] &= (x.y)\\
car[(x.y)] &= x\\
cdr[(x.y)] &= y\\
atom[x] &= \begin{cases}\text{T if } x \text{ is an atom}\\ \text{NIL otherwise}\end{cases}\\
eq[x, y] &= \begin{cases}\text{T if } x \text{ and } y \text{ are same atom}\\ \text{NIL otherwise}\end{cases}
\end{aligned}
$$

(We will also use a few contractions to aid readability. For example, $car[cdr[cdr[\ldots]]]$ will be shortened to $caddr[\ldots]$.)

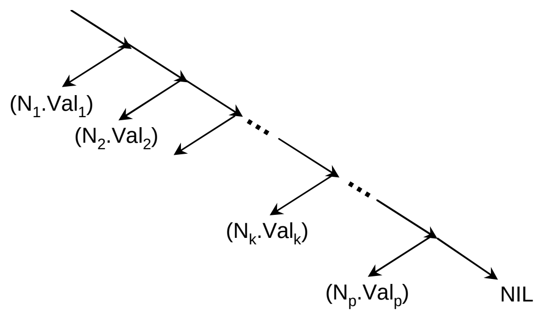
For the semantics of M-notation programs, we choose to work with a call-by-value language: all calls to primitive functions and user-defined functions are *strict*, with the exception of the **if-then-else** primitive, which is strict in the first position, but not in the second or third positions. Using such a language, we will define different interpreters for representation-notation programs, including call-by-value, call-by-name, and call-by-need.

3 Differences from the Programming Assignment about Interpreters

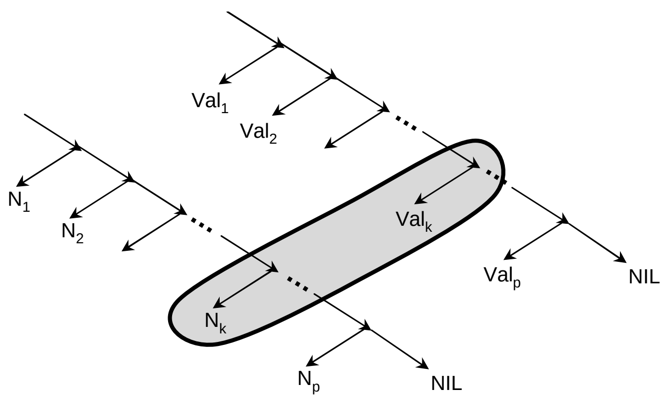
There will be a programming assignment in which, using ML as the meta-language, you will write different interpreters for Scheme0, including ones that implement call-by-value, call-by-name, and call-by-need. In the material presented in class, we will be diverging a little from what you have to do in the programming assignment. In particular, (i) the representations of environments will differ; (ii) the way closures are represented will differ (the assignment uses THUNK tags in the representations of closures); and (iii) in class we will present dynamic scoping as opposed to static scoping.

The difference in the treatment of scoping is a consequence of how we choose to set up the environment at function calls.

In the lecture, an environment is represented as a list of (name,value) pairs:



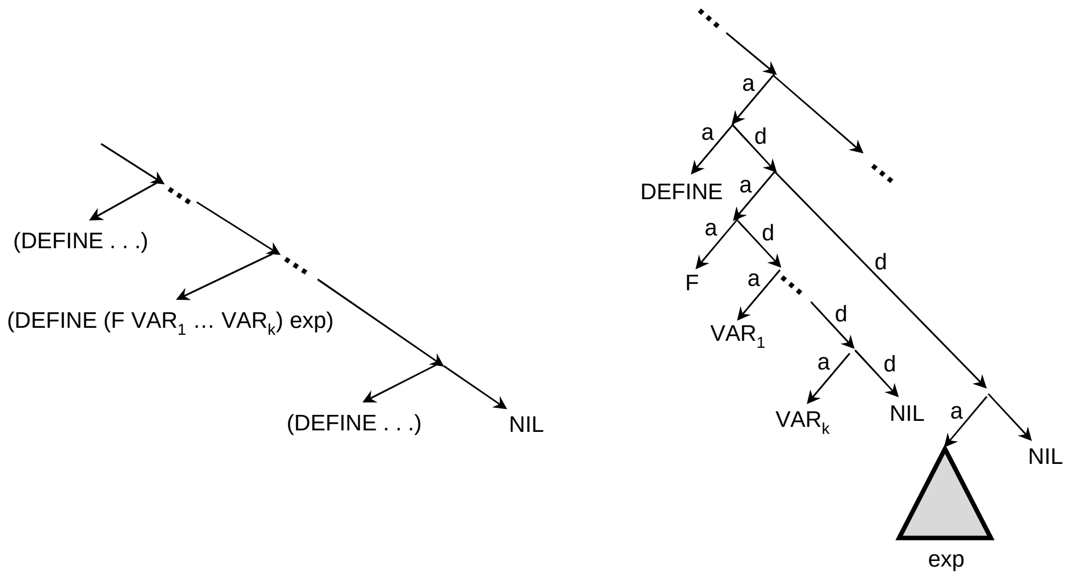
In contrast, in the programming assignment, you will represent an environment using "parallel lists": a name list and a value list that have the same length:



We will need the following utility function in the interpreters to allow the interpreter to retrieve information from the environment:

```
lookup[f, list] ← if null[list] then NIL
                  else if eq[f, caadar[list]] then car[list]
                  else lookup[f, cdr[list]]
```

To understand the access expression "caadar[list]", consider the following depiction of the environment:



4 Call-By-Value Interpreter

We can give representation-notation programs call-by-value semantics using the following interpreter:

```
eval[exp, env] ← cases exp
                    NIL : NIL
                    T : T
                    $n$ : $n$        // $n$ is a number
                    (QUOTE a) : a
                    Z : $V_Z$, where $((VAR_1\ .VAL_1)\ \ldots\ (\ Z\ .\ V_Z)\ \ldots\ (\ VAR_N\ .\ VAL_N)) = env$
```

```
(IF $e_1$ $e_2$ $e_3$) : if eval[$e_1$, env] then eval[$e_2$, env]
                         else eval[$e_3$, env]
(OP $e_1$ ... en) : perform_op[OP, eval[$e_1$, env], ..., eval[$e_n$,env]]
(CALL F $e_1$, ..., $e_n$) : eval[exp$'$, env$'$],where
        (DEFINE (F $x_1$ ... $x_n$) exp$'$) = lookup[$F_1$, env]
        $v'_j$ = eval[$e_j$, env]   for $j = 1 \ldots n$
        env$'$ = append[list[($x_1$ . $v'_1$) ... ($x_n$ . $v'_n$)], env]
```

For instance, consider the following program:

```
foo[x] ← 1/0
second[x, y] ← y
```

If we invoke the interpreter by calling

eval[(CALL SECOND (FOO (QUOTE B), (QUOTE A)), program]

it would return $\perp$ because of the division-by-0 performed when the actual parameter of FOO is evaluated.

5 Call-By-Name Interpreter

The ideas used in the interpreter that gives representation-notation programs call-by-name semantics are as follows:

1. Do not evaluate actuals.
2. Instead, pair the formals with the environment by creating "closures". A closure has the form (exp . env).
3. Invoke eval[exp, env] when we need to resolve (exp . env).

Thus, an environment will consist of "triples" of the form (name . (exp . env)).

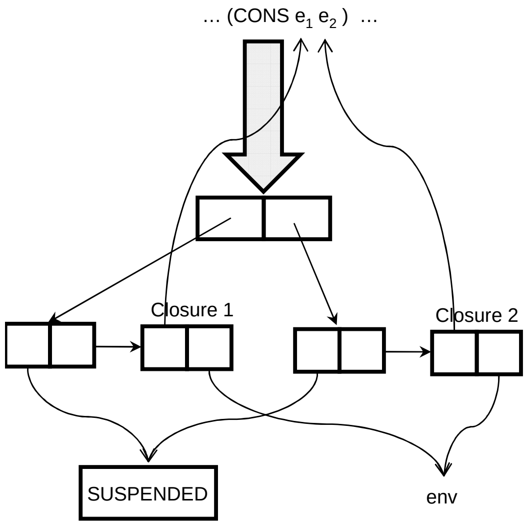
Compared to the call-by-value interpreter, only the cases for CALL and Variable Z need to change, the rest remain the same:

```
eval[exp, env] ← cases exp
        ...
        Z : force[$\text{Closure}_Z$], where
                (($\text{VAR}_1$ . $\text{Closure}_1$) ... (Z . $\text{Closure}_Z$) ... ($\text{VAR}_n$ . $\text{Closure}_n$)) = env
          and
                force[p] ← eval[car[p], cdr[p]]
        (CALL F $e_1$, ..., $e_n$) : eval[exp$'$, env$'$], where
                (DEFINE (F $x_1$ ... $x_n$) exp$'$) = lookup[$F_1$, env]
                env$'$ = append[list[($x_1$ . ($e_1$ . env)) ... ($x_n$ . ($e_n$ .env))], env]
```

6 A Lazy-List (or "Suspending-Cons") Interpreter

One advantage of the call-by-name interpreter is that it permits us to evaluate programs that manipulate infinite data structures. However, if we are just interested in infinite data structures, less drastic measures can be used. In particular, rather than creating a closure for each actual parameter for every kind of call, we can merely create such structures for the parameters of CONS operations. In other words, we can create a language that is able to manipulate infinite data structures by having an interpreter that uses closures much more parsimoniously than the call-by-name interpreter of §5.

In the suspending-cons interpreter given below, each CONS expression results in the creation of two "suspensions", which are not evaluated until the need arises. These structures are depicted below:



In addition, we have to modify the cases for CAR and CDR expressions to watch out for the atom "SUSPENDED", which tags each component of a suspended cons-expression.

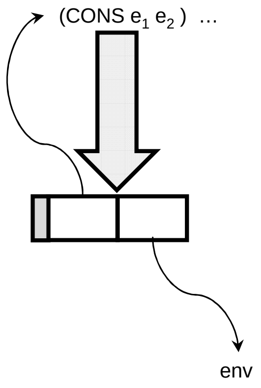
6.1 A Call-By-Need Version of the Suspending-Cons Interpreter

Consider a function in which a formal parameter (say 'A') is used multiple times. The call-by-name interpreter defined above will re-evaluate the expression to which 'A' is bound each time 'A' is encountered. The call-by-need version of the suspending-cons interpreter can be seen as the memoizing version of the call-by-name interpreter. Memoization allows it to avoid having to redo the same work when a formal parameter is accessed a second (or subsequent) time.

```
eval[exp, env] ← cases exp
                    (CONS $e_1$, $e_2$) : cons[suspend[$e_1$, env], suspend[$e_2$, env]], where
                            suspend[exp, env] ← cons[SUSPENDED, cons[$exp_1$, env]]
                    (CAR e) : first[v]
                    (CDR e) : rest[v]
                    where,
                         v = eval[e, env]
                         first[cell] ← **if** suspended[car[cell]]
                                          **then** car[coercecar[cell]]
                                       **else** car[cell]
                         rest[cell] ← **if** suspended[cdr[cell]]
                                         **then** cdr[coercecdr[cell]]
                                      **else** car[cell]
                         suspended[v] ← **if** atom[v]
                                              **then** NIL
                                           **else if** null[v]
                                              **then** NIL
                                           **else if** eq[car[v], SUSPEND]
                                              **then** T
                                           **else** NIL
                         coercecar[cell] ← rplaca[cell, eval[cadar[cell], cddar[cell]]]
                         coercecdr[cell] ← rplacd[cell, eval[caddr[cell], cdddr[cell]]]
```

6.2 A Space-Efficient Implementation of the Suspending-Cons Interpreter

As presented above, each suspended CONS causes five cons-cells to be allocated, and thus the interpreter is rather profligate in its use of storage. However, if we are able to steal one bit per cons-cell, then it is possible to create an alternative version in which each suspended CONS causes just a single cons-cell to be allocated. Given an expression (CONS E1 E2) to be evaluated in environment env, we need to freeze E1, E2, and env. At first, it seems like we need at least three new locations to store this information, which is one location too many for a single cons-cell. However, the expression (CONS E1 E2) of the program itself is already a cons-cell from which both E1 and E2 can be accessed. Thus, by allocating a single cons-cell (marked with a "Suspended" bit) in which the car-field points to (CONS E1 E2) and the cdr-field points to env, the interpreter can get by without having to create any other cells.



However, for this to work we must now coerce *both* halves of a suspended cons-cell whenever *either* a CAR expression or CDR expression is encountered.

```
coerce[cell] ← prog[replaca[cell,eval[cadar[cell],cdr[cell]],
                    replacd[cell,eval[caddar[cell],cdr[cell]],
                    flip the value of the bit that labels cell]
```

Consequently, an evaluation that creates and traverses (part of) an infinite data structure will elaborate a bit more of the visited part of the structure compared to the call-by-name version. That is, the space-efficient version has to elaborate the details of the structure that are one step away from the "spine" that is traversed; the less space-efficient version would just have to materialize the nodes along the spine. On the other hand, the space-efficient version gets away with using only one cons-cell per CONS construct encountered, which is the same amount of space allocated per CONS construct by the call-by-value interpreter.

7 Static versus Dynamic Scoping

The interpreters given above implement dynamic scoping: there is a single environment, and the new bindings established at a function call are appended to the beginning of the environment. Any use of a free variable will be looked up in the environment; if a binding for it happens to exist, the associated value will be used.

To change the behavior to use static scoping, we introduce a separate environment "prog" to hold the list of functions in the program, and change function calls to just evaluate the function body with an environment of variable bindings that consists solely of bindings for the function's formal parameters. For instance, the modified cases of the call-by-value interpreter are as follows:

```
eval[exp, env, prog] ← cases exp
                              . . .
                              (IF e$_1$ e$_2$ e$_3$): if eval[e$_1$, env, prog] then eval[e$_2$, env, prog]
                                                  else eval[e$_3$, env, prog]
                              (CALL F e$_1$...e$_n$): eval[exp$'$, env$'$, prog], where
                                   (DEFINE (F x$_1$...x$_n$) exp$'$) = lookup[f, prog]
                                   v$'_j$ = eval[e$_j$, env, prog]   for $j = 1 \ldots n$
                                   env$'$ = list[(x$_1$.v$'_1$), ...,(x$_n$.v$'_n$)]
                              (OP e$_1$...e$_n$): perform_op[OP, eval[e$_1$, env, prog], ...,eval[e$_n$, env, prog]]
```

References

[1] M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *Comp. Construct.*, 1992.

[2] P. Wadler. Efficient compilation of pattern matching. In *The Implementation of Functional Programming Languages*, chapter 5. Prentice-Hall, Englewood Cliffs, NJ, 1987. (See http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/.).