Lastref information provides just enough information to disentangle the multiple environments that can be created over a given variable. If we had lastref information, we could, for example, correctly perform type inference. Deriving lastref information is referred to as *environment flow analysis.*

A complete treatment of the environment problem, and techniques for computing the lastref function are beyond the scope of this paper. The next paper in this flow analysis series, *Environment Flow Analysis in Scheme*, will treat the environment problem, showing its solution, and the application of the lastref function to performing general flow analysis, including type inference as a demonstration example.

8.3 Mathematics

The mathematics of Scheme control, environment, and data flow analysis is captured by abstract semantic interpretations [Cousot] [Hudak2]. This will be the topic of another paper.

9 Acknowledgements

I would like to thank my advisor, Peter Lee, for carefully reviewing several drafts of this paper.

Notes

{Note Difficulties with Binding}

For all its pleasant properties, Lisp binding does introduce a problem that surfaces when we attempt to find applications for flow analysis-derived information, namely that lexical scoping coupled with "upward" functions gives the programmer very tight control over environments. This can frustrate optimising techniques involving code motion, where we might decide we would like to move a computation to a given control point p, only to find that the variables appearing in the computation are not available in the environment obtaining at p.

This problem does not arise with Algol-like languages, since their internal representations are typically close to assembly language. The environment is a single, large flat space, hence a given variable is visible over the entire body of code. Some further consequences of this difference are discussed in subsection 8.2.

{Note CPS and Triples}

CPS looks a lot like triples: generally, the code is broken down into primitive operations, which take the variables or constants they are applied to, and a continuation that specifies the variable to receive the computed value. CPS differs from triples in the following ways:

1. The continuation actually serves two roles. (1) It specifies the the variable to receive the value computed by the primitive operation. (2) It specifies where the control point will transfer to after executing the primitive operation. In triples this is split out.

2. A triple side-effects its target. A continuation binds its variable.

3. Continuations have runtime definitions; triples can be completely determined at compile time.

References

[Dragon] Aho, Ullman. *Principles of Compiler Design.* Addison-Wesley (1977).

[Hecht] Hecht, Matthew S. *Data Flow Analysis of Computer Programs.* American Elsevier (New York, 1977).

[R3-Report] J. Rees & W. Clinger, Ed.. "The *Revised*[3] Report on the Algorithmic Language Scheme." *SIGPLAN Notices* 21(12) (Dec. 1986), pp. 37–79.

[Declarative] Steele, Guy L. *Lambda: The Ultimate Declarative*. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Rabbit] Guy L. Steele. *Rabbit: A Compiler for Scheme.* AI-TR-474. MIT AI Lab (Cambridge, May 1978).

[ORBIT] Kranz, David, *et al.* "Orbit: An Optimizing Compiler for Scheme." *Proceedings of SIGPLAN '86 Symposium on Compiler Construction* (June 1986), pp. 219–233.

[LetS] Waters, Richard C. *LETS: an Expressional Loop Notation.* AI Memo 680. MIT AI Lab (Cambridge, October 1982).

[YLoop] Online documentation for the T3 implementation of the `Yloop` package is distributed by its current maintainer: Prof. Chris Riesbeck, Yale CS Dept. (`riesbeck@yale.arpa`).

[Hudak1] Hudak, Paul. "A Semantic Model of Reference Counting and its Abstraction." *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (August 1986).

[Hudak2] Hudak, Paul. *Collecting Interpretations of Expressions (Preliminary Version).* Research Report YALEU/DCS/RR-497. Yale University (August 1986).

[Cousot] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." *4th ACM Symposium on Principles of Programming Languages* (1977), pp. 238–252.

8 Limitations and Extensions

This paper is the first in a series intended to develop general flow analysis optimisations for Scheme-like languages. The basic control flow analysis technique of section 4 can be developed in many directions. In this section, I will sketch several of these directions, which are beyond the scope of this paper, and will be published in future reports.

8.1 First Order Closures: An Improvement

Control flow analysis attempts to deal with infinite structures by approximation. We would like to flow around *functions*. But the set of functions that a given program can produce is infinite. Consider the following infinite loop:

```
(labels ((f (lambda (g)
              (f (lambda ()
                   (+ (g) (g)))))))
   (f (lambda () 1)))
```

g is bound to the infinite set of functions $\{f(x) = 2^n | n \geq 0\}$

So, in general, there is no way to perfectly answer the question "what are all the functions called from a given call site?" The analysis technique presented in section 4 uses the approximation that identifies all functions that have the same "code," *i.e.* the functions that are closed over the same lambda expression.

We can use finer grained approximations, expending more work to gain more information. To borrow a technique from [Hudak1], we can track multiple closures over the same lambda. Since it's clear that in the real lambda semantics, a finite program can give rise to unbounded numbers of closures, we must identify some real closures together.

In the zeroth order control flow analysis (0CFA) presented in section 4, we identified all closures over a lambda together. In the first order case (1CFA), the contour created by calling a lambda from call point c_1 is distinguished from the contour created by calling that lambda from call point c_2. Closures over these two contours are distinct.

A full treatment of 1CFA will be found in the forthcoming CMU tech report[1] based on this conference paper.

8.2 Environment Flow Analysis

Although control flow analysis provides enough information to perform some traditional flow analysis optimisations on Scheme code, *e.g.* induction variable elimination, it is not sufficient to solve the full range of flow analysis problems. Why is this?

Traditional flow analysis techniques assume a single, flat environment over the program text. New computed values are communicated from one portion of the program to another by side-effecting and referencing global variables. Thus, any two references to the same variable refer to the same binding of that variable. This is, for instance, the model provided by assembly code.

Our technique, however, uses a different intermediate representation, CPS applicative-order λ-calculus. Here, the situation

---

[1]Due to space limits, the description of Useless Variable Elimination and the algorithm for finding BIV families are also deferred to the tech report.

is not so simple. The indefinite extent of variable bindings implies that we can have simultaneous *multiple* bindings of the *same* variable. For instance, consider the following perverse example:

```
(let ((f (lambda (h x)
           (if (zero? x) (h)
                         (lambda () x))))))
   (f (f nil 3) 0))
```

The first call to f, (f nil 3) returns a closure over (lambda () x), in an environment [x/3]. This function is then used as an argument to the second, outer call to f, where it is bound to the variable h. Suppose we naively picked up information from the conditional (zero? x) test, and flowed $x = 0$ and $x \neq 0$ down the THEN arm and the ELSE arm of the if, respectively. This would work in the single flat environment model of assembler. In our example, however, we are undone by the power of the lambda. After ensuring that $x = 0$ in the THEN arm of the (if (zero? x)...) conditional, we jump off to h, and evaluate x, which inconveniently persists in evaluating to 3, not 0. The one variable has multiple simultaneously extant values. Confusing two bindings of the same variable can lead us to draw incorrect conclusions.

This problem prevents us from using plain control flow analysis to perform an important flow analysis optimisation for Lisp: type inference. We would like to perform deductions of the form:

```
;; references to x in f are int refs
(if (integer? x) (f) (g))
```

Unfortunately, as demonstrated above, we can't safely make such inferences.

What is missing is environment information. We need to know which references to a variable occur in the same environment. For example, consider the lambda expression:

```
(lambda (0:x)
  (cond ((zero? 1:x)
         (print 2:x) (* 3:x 4))
        (t (+ 4:x 7))))
```

There is no execution path of a program containing the above lambda such that reference 2:x occurs in the same environment as reference 4:x. On the other hand, in all execution paths, reference 2:x occurs in the same environment as reference 3:x.

We cannot do general data flow analysis unless we can untangle the multiple environments that variables can be bound in. This is not surprising. As is pointed out in [Declarative], lambda serves a dual semantic role of providing both control and environment structure. Control flow analysis has provided information about only one of these structures. We must also gather information about the relationships among the binding environments established during program execution.

The exact nature of the environment information we need is succinctly captured in the *lastref* function:

> For a reference *r* to variable *v*, what is *LR(r)*, the set of possible *immediately prior* references to the same binding of *v*?

```
(labels ((f (lambda (n) (if (< n 50)
                            (if (= (aref a n) n) (f (+ n 1))
                                (block (print (+ 4 (* 3 n)))
                                       (f (+ n 2)))))))))
  (f 0))
```
(a) Before

```
(labels ((f (lambda (n)
              (if (< n 50)
                  (if (= (aref a n) n) (+ n 1 f)
                      (* 3 n (lambda (m) (+ 4 m (lambda (p) (print p)
                                                            (+ n 2 f)))))))))))
  (f 0))
```
(b) Partial CPS conversion

```
(labels ((f (lambda (n 3n)
              (if (< n 50)
                  (if (= (aref a n) n)
                      (+ 3n 3 (lambda (3n')
                                (+ n 1 (lambda (n') (f n' 3n')))))
                      (+ 4 3n (lambda (p) (print p)
                                          (+ 3n 6 (lambda (3n%)
                                                    (+ n 2 (lambda (n%)
                                                             (f n% 3n%)))))))
                      )))))
  (f 0 0))
```
(c) IVE wave 1

```
(labels ((f (lambda (n 3n 3n+4)
  (if (< n 50)
      (if (= (aref a n) n)
          (+ 3n+4 3 (lambda (3n+4')
                      (+ 3n 3 (lambda (3n') (+ n 1 (lambda (n')
                                                     (f n' 3n' 3n+4')))))))
          (block
            (print 3n+4)
            (+ 3n+4 6 (lambda (3n+4%)
                        (+ 3n 6 (lambda (3n%) (+ n 2 (lambda (n%)
                                                       (f n% 3n% 3n+4%))))))))))))
  (f 0 0 4))
```
(d) IVE wave 2

```
(labels ((f (lambda (n 3n+4)
              (if (< n 50)
                  (if (= (aref a n) n)
                      (+ 3n+4 3 (lambda (3n+4') (+ n 1 (lambda (n') (f n' 3n+4')))))
                      (block
                        (print 3n+4)
                        (+ 3n+4 6 (lambda (3n+4%) (+ n 2 (lambda (n%)
                                                           (f n% 3n+4%)))))))))))
  (f 0 4))
```
(e) After

Figure 4: Example IVE application

As an example, consider induction variable elimination. Induction variable elimination is a technique used to transform array references inside loops into pointer incrementing operations. For example, suppose we have the following C routine:

```
int a[50][30];

example() {
    integer r, c;
    for(r=0; r<50; r++)
        for(c=0; c<30; c++)
            a[r][c] = 4;
}
```

The `example` function assigns 4 to all the elements of array a. The array reference `a[r][c]` on a byte addressed machine is equivalent to the address computation `a+4*(c+30*r)`. This calculation requires two multiplications and two additions. By replacing the array indices with a pointer variable, we can remove the entire address calculation:

```
example() {
    integer *ptr;
    for(ptr=a; ptr<a+1500; ptr++)
        *ptr = 4;
}
```

Induction variable elimination is a technique for automatically performing the above transformation. The Lisp variant uses control flow analysis.

A *basic induction variable* (BIV) family is a set of variables $B = \{v_1, \ldots, v_n\}$ obeying the following constraints:

1. Every call site that calls one of the v_i's lambda expression may only call one lambda expression. In terms of the Defs and L functions given earlier, this is equivalent to stating that for all call sites c such that (lambda $(\ldots v_i \ldots)$ $\ldots$) $\in L(c)$ for some v_i, $L(c)$ must be a singleton set:

   $\forall$call site's c, $v_i \in B$

   (lambda $(\ldots v_i \ldots)$ $\ldots$) $\in L(c) \Rightarrow |L(c)| = 1$

2. Each v_i may only have definitions of the form b (constant b), and $a + v_j$ ($v_j \in B$, constant a). This corresponds to stating that v_i's lambda must be one of:

   - Called from a call site that simply binds v_i to a constant:
     ((lambda (x vi z) ...) foo 3 bar).

   - The continuation of a + call, with v_j and a constant for addends:
     (+ vj a (lambda (vi) ...)).

   - The continuation of a – call, with v_j and a constant for subtrahend and minuend:
     (- vj a (lambda (vi) ...)).

   - Called from a call site that simply binds v_i to some v_j, *e.g.*:
     ((lambda (x vi z) ...) foo vj bar)
     (This is a special case of $a = 0$).

Given the control flow information, it is fairly simple to compute sets of variables satisfying these constraints.

A *dependent induction variable* (DIV) family is a set of variables $B' = \{w_i\}$ together with a function $f(n) = a + b * n$ (a,b constant) and a BIV family $B = \{v_i\}$ such that:

- For each w_i, every definition of w_i is $w_i = f(v_j)$ for some $v_j \in B$.

We call the value $f(v_i)$ the *dependent value.*

We can introduce three sets of new variables, $\{z_i\}$, $\{x_i\}$, and $\{y_i\}$. The z_i and x_i track the dependent value, and the y_i are introduced as temporaries to help update the basic variable. In the following description, we use for illustration the following piece of Lisp code:

```
(+ v1 4 (lambda (v2)
   ^          ...        ^
   |        (* 3 v2|(lambda (w1) ...)))))
   |               |          ^
   +-------------+    DIV w1 defined
   BIVs v1 & v2          to be 3*v2.
```

with associated function $f(n) = 3 * n$.

We perform the following transformations on the code:

- **Introduce z_i into the v_i lambda expression:**
  Modify each lambda that binds one of the v_i to simultaneously bind the corresponding value $f(v_i)$ to z_i.
  Binding site (lambda (v2) ...) becomes:
  (lambda (v2 z2) ...)

- **Introduce code to compute $z_j = f(v_j)$ from $z_i = f(v_i)$ when v_j is computed from v_i**
  All call sites who have continuations binding one of the v_i (*i.e.* call sites that step the BIV family) are modified to update the corresponding z_i value. Temporary variables x_i and y_i are used to serially compute the new values.
  Call site (+ v1 4 k) becomes:

  ```
  (+ z1 12 (lambda (x2)
              (+ v1 4 (lambda (y2)
                         (k y2 x2)))))
  ```

- **Replace computation of DIV w_i from v_j with simple binding:**
  Since $z_j = f(v_j)$, we can remove the computation of w_j from v_j, and simply bind w_j to z_j. Dependent variable computation (* v2 3 (lambda (w1) ...)) becomes: ((lambda (w1) ...) z2) and z_2 can be β substituted for w_1.

Notice that $\{w_i\} \cup \{z_i\} \cup \{x_i\}$ now form a new BIV family, and may trigger off new applications.

For example, consider the loop of figure 4(a). It is converted to the partial CPS of (b). Now, $\{n\}$ is a BIV family, and $\{m\}$ is a DIV family dependent on $\{n\}$, with $f(n) = 3n$. A wave of IVE transformation gives us the optimised result of (c). Analysis of (c) reveals that $\{3n, 3n', 3n\%\}$ is a BIV family, and $\{p\}$ is a DIV family, with $f(n) = 4 + n$. Another wave of IVE gives us the (d). If we examine (d), we notice that 3n, 3n', and 3n% are never used, except to compute values for each other. Another analysis using control flow information, Useless Variable Elimination spots these useless variables; they can be removed, leaving the optimised result of (e).

5 Two Examples

Consider the program

```
(lambda () (if 5 (+ 3 4) (- 1 2)))
```

which evaluates to 7. In CPS form, this is:

```
(lambda (k1) (%if 5 (lambda () (+ 3 4 k1))
                     (lambda () (- 1 2 k1))))
```

Labeling all the lambda expressions and call sites for reference, we get:

```
l1:(lambda (k1)
     c1:(%if 5 l2:(lambda () c2:(+ 3 4 k1))
              l3:(lambda () c3:(- 1 2 k1))))
```

ESCAPED is initially $\{\text{XLAMBDA}, l_1\}$. Our escaped function rules tell us that l_1 can be called from XCALL, or that $l_1 \in L(\text{XCALL})$, and that $\{l_1, \text{XLAMBDA}\} \subset \text{Defs}(k_1)$. The propagation condition for %if gives us that %if's first internal call site, $ic^1_{\%if}$, calls l_2, *i.e.* $l_2 \in L(ic^1_{\%if})$, and %if's second internal call site, $ic^2_{\%if}$, calls l_3, *i.e.* $l_3 \in L(ic^2_{\%if})$. The propagation condition for + gives us that ic_+ contains $\text{Defs}(k_1) = \{l_1, \text{XLAMBDA}\}$, or that +'s internal continuation call can transfer control to l_1 or the XLAMBDA. Similarly, we derive for − that $L(ic_-)$ contains $\{l_1, \text{XLAMBDA}\}$. This completes the control flow analysis for the program.

Consider the infinite loop:

```
(labels ((f (lambda (n) (f n))))
  (f 5))
```

With top-level continuation *k*, this is CPSised into:

```
(lambda (*k*)
  (Y (lambda (ignore-b f k1)
       (k1 (lambda (k2) (f 5 k2))
           (lambda (n k3) (f n k3))))))
  (lambda (b ignore-f) (b *k*)))
```

Labelling all the call sites for reference:

```
(lambda (*k*)
  c1:(Y (lambda (ignore-b f k1)
          c2:(k1 (lambda (k2) c3:(f 5 k2))
                 (lambda (n k3) c4:(f n k3))))))
      (lambda (b ignore-f) c5:(b *k*)))
```

Y's internal call site is labelled ic_Y. After performing the propagations, we get:

$$
\begin{aligned}
L(\text{XCALL}) &= \{(\texttt{lambda (*k*) ...}), \text{XLAMBDA}\} \\
L(c_1) &= \{\text{Y}\} \\
L(ic_Y) &= \{(\texttt{lambda (ignore-b f k1) ...})\} \\
L(c_2) &= \{(\texttt{lambda (b ignore-f) ...})\} \\
L(c_3) &= \{(\texttt{lambda (n k3) (f n k3)})\} \\
L(c_4) &= \{(\texttt{lambda (n k3) (f n k3)})\} \\
L(c_5) &= \{(\texttt{lambda (k2) (f 5 k2))})\}
\end{aligned}
$$

6 Side Effects

For the purposes of doing control flow analysis of Lisp, tracking side effects is not of primary importance. Since Lisp makes variable binding convenient and cheap, side effects occur relatively infrequently. (This is especially true of well written Lisp.) The pieces of Lisp we are most interested in flow analysing — inner loops — use side effects primarily for updating iteration variables, which are rarely functional values. The control flow of inner loops is usually explicitly written out, and tends not to use functions as first class citizens.

To further editorialise, I believe that the updating of loop variables is a task best left to loop packages such as Waters' LetS [LetS] or the Yale Loop [YLoop], where the actual updating technique can be left to the macro writer to implement efficiently, and ignored by the application programmer. Even the standard Common Lisp and Scheme looping construct, do, permits a binding interpretation of its iteration variable update semantics.

For these reasons, we can afford to settle for a solution that is merely correct, without being very informative. Therefore, the control flow analysis presented here uses a very weak technique to deal with side effects. The CPS transformation discussed in section 3 removes all side effects to variables. Only side effects to data structures are allowed. All side effects and accesses to data structures are performed by primops. Among these primops are cons, rplaca, rplacd, car, cdr, vref, and vset. We divide these into two sets: *data stashers*, and *data fetchers*. Stashers, such as cons, rplaca, rplacd, and vset, take functions and tuck them away into some data structure. Fetchers — car, cdr, vref — fetch functions from some data structure.

The analysis takes the simple view that once a function is stashed into a data structure, it has escaped from the analyser's ability to track it. It could potentially be the value of any data structure access, or escape outside the program to the ESCAPED set. Thus we have two rules for side effects:

1. Any lambda passed to a stasher primop is included in the ESCAPED set.

$$(\texttt{cons } a \ d \ kont) \Rightarrow \qquad \text{[S-1]}$$
$$\text{Defs}(a), \text{Defs}(d) \subset \text{ESCAPED}$$
$$(\texttt{vset } vec \ index \ val \ kont) \Rightarrow \qquad \text{[S-2]}$$
$$\text{Defs}(val) \subset \text{ESCAPED}$$
$$etc.$$

2. A fetcher primop can pass any ESCAPED function to its continuation:

$$(fetcher \ \ldots \ cont) \Rightarrow \qquad \text{[F-1]}$$
$$\forall (\texttt{lambda } (x) \ \ldots) \in \text{Defs}(cont),$$
$$\text{ESCAPED} \subset \text{Defs}(x)$$

In a sense, stashers and fetchers act as black and white holes: values disappear into stasher primops and pop out at fetcher primops.

7 Induction Variable Elimination: An Application

Once we've performed control flow analysis on a program, we can use the information gained to do other data flow analyses. Control flow analysis by itself does not provide enough information to do all the data flow problems we might desire — the limitations of control flow analysis are discussed in the section 8 — but we can still solve some useful problems.

internal call site, which marks the call to the primop's continuation. %if has two internal call sites, one for the consequent continuation, and one for the alternate continuation.

Let $ic_{p,c}^j$ be the jth internal call site for the primop p called at call site c.

An ordinary primop — *e.g.* +, cons, aref — takes its single continuation as its last argument. Any lambda which that last argument could evaluate to, then, can be called from the internal call site of the primop. That is, in call c :(*primop arg$_1$...cont*),

$$\text{Defs}(cont) \subset L(ic_{primop,c}^1) \qquad \text{[PRIM]}$$

%if is a special primop, in that it takes two continuations as arguments, either of which can be branched to from inside the %if. %if has two internal call sites, the first for the *consequent* continuations, and the second for the *alternate* continuations. So there are two propagation conditions for an occurence of %if, c :(%if *pred cons alt*). Any lambda the consequent continuation can evaluate to can be called from the first internal call site:

$$\text{Defs}(cons) \subset L(ic_{\%if,c}^1) \qquad \text{[IF-1]}$$

Any lambda the alternate continuation can evaluate to can be called from the second internal call site:

$$\text{Defs}(alt) \subset L(ic_{\%if,c}^2) \qquad \text{[IF-2]}$$

The propagation condition for the Y operator is more complicated. The Y primop is not available to the user; it is only introduced during CPS conversion of labels expressions. The CPS transformation for labels expressions turns

```
(labels ((f f-lambda)
         (g g-lambda))
   body)
```

with continuation kont into:

```
(Y (lambda (huaskt f g c)
     (c (lambda (k) body)
        f-lambda
        g-lambda)))
   (lambda (bodyfun hunoz hukairz)
     (bodyfun kont)))
```

where hunoz, hukairz, and huaskt are unreferenced — ignored dummy variables.

Since Y is only introduced into the program text in a controlled fashion, we can make particular assumptions about the syntax of its arguments. In particular, we can assume that *f-lambda* and *g-lambda* are lambda expressions, and not variables. This restriction can be relaxed with the addition of a small amount of extra machinery.

We call the functional which is Y's second argument the *fixpointer*. Y's propagation condition has three parts. For an occurence of the Y primop:

c :(Y (lambda (v_1... v_n k) (k f_1... f_n)) *cont*)

1. Y calls the fixpointer from its internal continuation call site:

$$(\text{lambda } (v_1...v_n \ k) \ ...) \in L(ic_{Y,c}) \qquad \text{[Y-1]}$$

2. Y's continuation is passed to the fixpointer as its continuation argument:

$$\text{Defs}(cont) \subset \text{Defs}(k) \qquad \text{[Y-2]}$$

3. Extract the args to the call to k, *i.e.* the definitions f_i of the label functions v_i, and for each lambda f_i, pass $\{f_i\}$ in as the fixpointer's ith argument:

$$f_i \in \text{Defs}(v_i) \qquad \text{[Y-3]}$$

4.3 Handling External References

Besides call sites, lambda expressions and primops, there are two other special syntactic values we must represent in our analysis. We need a way to represent unknown functions that are passed into our program from the outside world at run time. In addition, we need a call site to correspond to calls to functions that happen external to the program text. For example, suppose we have the following fragment in our program

```
(foo (lambda (k) (k 7))
```

If foo is free over the program, then in general we have no idea at compile time what functions foo could be bound to at run time. The call to foo is essentially an escape from the program to the outside world, and we must record this fact. We do this by including XLAMBDA, the *external lambda*, in Defs[foo]. Further, since (lambda (k) (k 7)) is passed out of the program to external routines, it can be called from outside the program. This is represented with a special call site, XCALL, the *external call*. We record that (lambda (k) (k 7)) is in L(XCALL).

Functions such as (lambda (k) (k 7)) in the above example that are passed to the external lambda have escaped to program text unavailable at compile time, and so can be called in arbitrary, unobservable ways. They have escaped close scrutiny, and in the face of limited information, we are forced simply to assume the worst. We maintain a set ESCAPED of escaped functions, which initially contains XLAMBDA and the top level lambda of the program. The rules for the external call, the external lambda and escaped functions are simple:

1. Any escaped function can be called from the external call:

$$\text{ESCAPED} \subset L(\text{XCALL}) \qquad \text{[X-1]}$$

2. Any escaped function may be applied to any escaped function: $\forall l \in$ ESCAPED of the form (lambda ($...v_i...$) $...$),

$$\text{ESCAPED} \subset \text{Defs}(v_i) \qquad \text{[X-2]}$$

This provides very weak information, but for external functions whose properties are completely unknown at compile time, it is the best we can do. (On the other hand, many external functions, *e.g.* print, or length, have well known properties. For instance, both print and length only call their continuations. Thus, their continuations do not properly escape. Nor are their continuations passed escaped functions as arguments. It is straightforward to extend the analysis presented here to utilise this stronger information.)

```
(lambda (n k)   ; Call K on N!          (lambda (n k)   ; Call K on N!
  ;; Y calls continuation (LAMBDA (G)...) on      (y (lambda (f c)
  ;; fixpoint of (LAMBDA (F C) ...).                 (c (lambda (m a k1)
  ;; (LAMBDA (M K1) is factorial code.                    (test-zero? m
  (y (lambda (f c)                                             (lambda () (k1 a))
       (c (lambda (m k1)                                       (lambda ()
            (test-zero? m                                        (* a m (lambda (a1)
                (lambda () (k1 1))                                         (- m 1 (lambda (m1)
                (lambda ()                                                         (f m1 a1 k1))
                  (- m 1 (lambda (m1)                                      )))))))))
                          (f m1 (lambda (a)          (lambda (g) (g n 1 k))))
                                  (* a m k1))
                )))))))))
     (lambda (g) (g n k))))
```

(a) Recursive factorial	*(b) Iterative factorial*

Figure 3: Two factorial functions using the CPS Y operator

As an initial approximation, let's discuss control flow analysis for a Scheme that does not allow side effects, even side effects to data structures. This allows us to avoid worrying about functions that "escape" into data structures, to emerge who-knows-where. Later in the paper, we will patch our side-effectless solution up to include side-effects.

The point of using CPS Lisp for an intermediate representation is that all transfers of control are represented by function call. Thus the *control flow problem* is defined by the following question:

> For each call site c in program P, what is the set $L(c)$ of lambda expressions that c could be a call to? *I.e.*, if there is a possible execution of P such that lambda l is called from call site c, then l must be an element of $L(c)$.

The definition of the problem does not define a unique function L. The trivial solution is the function $L(c) =$ AllLambdas, *i.e.* the conclusion that all lambdas can be reached from any call site. What we want is the tightest possible L. It is possible, in fact, to construct algorithms that compute the smallest L function for a given program. The catch lies in determining when to halt the algorithm. In general, then, we must be content with an approximation to the optimal solution.

Let ARGS be the set of arguments to calls in program P, and LAMBDAS be the set of lambda expressions in P. That is, ARGS is the set of variables and lambda expressions in P (we assume variables names are unique here, *i.e.* that variables have been α-converted to unique names.) We define a function Defs : ARGS $\rightarrow \mathcal{P}$(LAMBDAS). Defs(a) gives all the lambda expressions that a could possibly evaluate to. That is,

$$\text{Defs}[(\texttt{lambda ...})] = \{(\texttt{lambda ...})\}$$
$$\text{Defs}[primop] = \{primop\}$$
$$\text{Defs}[v] = \{l \mid v \text{ bound to lambda } l\}$$

L is trivially determined from Defs. In a call $c : (f\, a_1 \ldots a_n)$, $L(c)$ is just Defs(f).

4.1 Handling Lambda

Defs can be given with a recursive definition: In a call $c : (f \ldots a_i \ldots)$, $\forall l \in$ Defs(f) of the form (lambda (... v_i...) ...),

$$\text{Defs}(a_i) \subset \text{Defs}(v_i) \qquad \text{[LAM]}$$

I.e. if lambda l can be called from call site c, then l's ith variable can evaluate to any of the lambdas that c's ith argument can evaluate to.

What we are doing here is willfully confusing closures with lambda expressions. A lambda expression is not a function; it must be paired with an environment to produce a function. When we speak of "calling a lambda expression l," we really mean: "calling some function f which is a closure of lambda expression l." An alternate view is that we are reducing a potentially infinite set of functions to a finite set by 'folding' together the environments of all functions constructed from the same lambda expression. This issue will be dealt with in detail in a later section.

4.2 Handling Primops

It can be seen from the definition of Defs that we are flowing information about which lambdas are called from which call sites. But this is not the whole story. Not all function calls happen at call sites. Consider the fragment (+ a b (lambda (s) (f a s))). Where is (lambda (s) (f a s)) called from? It is called from the innards of the + primop; there is no corresponding call site in the program syntax to mark this. We need to endow primops with special *internal call sites* to mark calls to functions that happen internal to the primop. Different primops have different calling behavior with respect to their arguments: + calls its third argument only. %if calls its second and third arguments. Y has even more complex behavior. So we model each primop specially.

For each call to each primop $c : (primop\, arg_1 \ldots)$, we associate a related set of internal call sites $ic_1, \ldots, ic_n$ for the primop's use. Most normal primops, *e.g.* +, have a single

– **Args:**
An argument is a lambda expression, a variable reference, or a constant.

- **Primitive Operations:**
A primop denotes some primitive function, and is given by a predefined identifier, *e.g.*, +, %if.

- **Variables:**
Variables have the standard identifier syntax.

- **Constants:**
Constants have no functional interpretation, and so have syntax and semantics of little relevance to our analysis. I use integers, represented by base 10 numerals, in my examples.

Note that this syntax specification relegates primops to second class status: they may only be called, not used as data, or passed around as arguments. This leads to no loss of generality, since everywhere one would like to use the + primop as data, for instance, one can instead use an equivalent lambda expression: (lambda (a b c) (+ a b c)).

The reason behind splitting out primops as second class is fairly operational: we view primops as being small, open-codeable types of objects. Calls to primops are where computation actually gets done. There are other formulations possible with first-class primops, with corresponding flow analytic solutions. Relegating primops to second class status simplifies the presentation of the analysis technique presented in this paper.

Note also that the definition of CPS Lisp implies that the only possible body a lambda expression can have is a function call. This is directly reflected in our syntax specification.

A restatement of the syntactic invariants in our representation:

- A lambda's body consists of a single function call.

- A function call's arguments may only be lambdas, variables, or constants. That is, nested calls of the form: (f (g a) (h b)) are *not allowed.*

Some Primops:

%if, test: As discussed above, %if is a primop taking three arguments: a boolean, and two continuations. If the boolean is true, the first continuation is called, otherwise the second continuation is called.

%if can have specialised test forms that take a non-boolean first argument, and perform some test on it, *e.g.* (test-zero? x f g) calls f if x is zero, otherwise g. (test-nil? y h k) calls h if y is nil, otherwise k.

Y: Y is the CPS version of the λcalculus "paradoxical combinator" fixpoint function. The CPS definition is tricky, and is best arrived at in stages. Consider the following use of the non-CPS fixpoint operator:

```
(Y (lambda (f)
     (lambda (n)
       (if (zero? n) 1
           (* n (f (- n 1)))))))
```

Y returns the fixpoint of its argument, *i.e.* that function f' such that (lambda (f) ...) applied to f' yields f'. However, if we convert (lambda (f) ...) to CPS notation, we get:

```
(lambda (f k)
  (k (lambda (n c)
       (test-zero? n
         (lambda () (c 1))
         (lambda ()
           (- n 1 (lambda (n1)
                    (f n1 (lambda (a)
                            (* a n c))
                   )))))))))
```

Note that our functional doesn't return a value (since no CPS function returns a value). Instead, it calls its continuation k on its computed value. So the "fixpoint" of our new, CPS functional is that function f' such that (lambda (f k) ...) applied to f' and some continuation c, calls c on f'. That is, calling the continuation on f' is equivalent to returning f'. We can generalise our notion of "fixpoint" to include groups of mutually recursive functions by allowing our functional to take more than one non-continuation argument, *i.e.* a "fixpoint" of some CPS function (lambda (f g h c) ...) is a collection of three functions, f', g', and h', such that if (lambda (f g h c) ...) is applied to f', g', h', and some continuation k, then f', g', and h' will be passed to the continuation k.

So our CPS version of the Y combinator looks like:

```
(Y (lambda (f g h k) (k f-definition
                         g-definition
                         h-definition))
   c)
```

The result of this example is to call continuation c on three values, the fixpoints f', g', and h' of (lambda (f g h k) ...).

Figure 3 shows two complete examples using the CPS Y operator. {Note CPS and Triples}

4   Control Flow Analysis: A Technique

Our intermediate representation defined, we can now proceed to develop a technique for deriving the control flow information present in a Scheme program. The solution to the dilemma of section 1 is to use a flow technique which will "bootstrap" the control flow graph into being. As is typical in flow analysis problems, our solution may err, as long as it errs *conservatively.* That is, it may introduce spurious edges into the control flow graph, but may not leave out edges representing transfers of control that could actually occur during execution of the program.

The intuition is that once we determine the control flow graph for the Lisp function, we can then use it to do other, standard data flow analyses. We refer to the problem of determining the control flow graph for the purposes of subsequent data flow analysis as *control flow analysis.* (This intuition will only be partially borne out. The limitations of control flow analysis will be discussed in section 8.)

"standard" Lisp is mapped into a much simpler, restricted subset of Lisp, which has the effect of greatly simplifying the analysis.

In Lisp, we must represent and deal with transfers of control caused by function calls. This is most important in the Scheme dialects [R3-Report], where lambda expressions occur with extreme frequency. In the interests of simplicity, then, we adopt a representation where *all* transfers of control — sequencing, looping, function call/return, conditional branching — are represented with the same mechanism: the tail recursive function call. This representation is called CPS, or *Continuation Passing Style*, and is treated at length in [Declarative].

CPS stands in contrast to the intermediate representation languages commonly chosen for traditional optimising compilers. These languages are conventionally some form of slightly cleaned-up assembly language: quads, three-address code, or triples. The disadvantage of such representations are their *ad hoc*, machine-specific, and low-level semantics. The alternative of CPS was first proposed by Steele in [Rabbit], and further explored by Kranz, *et al.* in [ORBIT]. The advantages of CPS lie in its appeal to the formal semantics of the λcalculus, and its representational simplicity.

CPS conversion can be referred to as the "hedgehog" approach, after a quotation by Archilocus. All control and environment structures are represented in CPS by lambda expressions and their application. After CPS conversion, the compiler need only know "one great thing" — how to compile lambdas very well. This approach has an interesting effect on the pragmatics of Scheme compilers. Basing the compiler on lambda expressions makes lambda expressions very cheap, and encourages the programmer to use them explicitly in his code. Since lambda is a very powerful construct, this is a considerable boon for the programmer.

In CPS, function calls are *tail recursive*. That is, they do not return; they are like GOTOs. If we are interested in the value computed by a function f of two values, we must pass f a third argument, the *continuation*. The continuation is a function; after f computes its value v, instead of "returning" the value v, it calls the continuation on v. Thus the continuation represents the control point which will be transferred to after the execution of f.

For example, if we wish to print the value computed by $(x + y) * (z + w)$, we do not write:

```
(print (* (+ x y) (+ z w)))
```

instead, we write:

```
(+ x y (lambda (xy)
        (+ z w (lambda (zw)
                (* xy zw print)))))
```

Here, the primitive operations — +, *, and print — are all redefined to take a continuation as an extra argument. The first + function calls its third argument, (lambda (xy) ...) on the sum of its first two arguments, x and y. Likewise, the second + function calls its third argument, (lambda (zw) ...) on the sum of its first two arguments, z and w. And the * function calls its third argument, the print function, on the product of its first two arguments, $x + y$, and $z + w$.

Continuation passing style has the following invariant:

- The only expressions that can appear as arguments to a function call are constants, variables, and lambda expressions.

Standard non-CPS Lisp can be easily transformed into an equivalent CPS program, so this representation carries no loss of generality. Once committed to CPS, we make further simplifications:

- **No special syntactic forms for conditional branch.**
  The semantics of primitive conditional branch is captured by the primitive function %if which takes one boolean argument, and two continuation arguments. If the boolean is true, the first continuation is called; otherwise, the second is called.

- **No special** labels **or** letrec **syntax for mutual recursion.**
  Mutual recursion is captured by the primitive function Y, which is the "paradoxical combinator" of the λcalculus.

- **Side effects to variables are not allowed.**
  We allow side effects to data structures only. Hence side effects are handled by primitive functions, and special syntax is not required.

Lisp code violating any of these restrictions is easily mapped into equivalent Lisp code preserving them, so they carry no loss of generality. In point of fact, the front end of the OR-BIT compiler [ORBIT] performs the transformation of standard Scheme code into the above representation as the first phase of compilation.

These restrictions leave us with a fairly simple language to deal with: There are only five syntactic entities: lambdas, variables, primops, constants and calls (fig. 2), and two basic semantic operations: abstraction and application.

- Lambda expressions: (lambda *var-set call*)

- Variable references: foo, bar, ...

- Constants: 3, "doghen", '(a 3 elt list),...

- Primitive Operations: +, %if, Y, ...

- Function calls: (*fun arg**)
  where *fun* is a lambda, var, or primop, and the *arg*s are lambdas, vars, or constants.

Figure 2: CPS language grammar

- **Lambda Expressions:**
  A lambda expression has the syntax (lambda *var-set call*), where *var-set* is a list of variables, *e.g.* (n m a), and *call* is a function call. A lambda expression denotes a function.

- **Function Calls:**
  A function call has the syntax (*fun* $arg_1 \ldots arg_n$) for $n \geq 0$. *fun* must be a function, and the arg_i must be args. [see below]

  - **Function:**
    A function is a lambda expression, a variable reference, or a primop.

developed for them are not applicable.

Dialects of Lisp can be contrasted with the Algol family in the following ways:

- **Binding versus assignment:**
  Both classes of language have the same two mechanisms for associating values with variables: parameter binding and variable assignment. However, there are differences in frequency of useage. Algol-family languages tend to encourage the use of assignment statements; Lisp languages tend to encourage binding.

- **Functions as first class citizens:**
  Functions in modern Lisps are data that can be passed as arguments to procedures, returned as values from function calls, stored into arrays, *etc.*.

Since traditional flow analysis techniques tend to concentrate on tracking assignment statements, it's clear that the Lisp emphasis on variable binding changes the complexion of the problem. Generally, however, variable binding is a simpler, weaker operation than the extremely powerful operation of side effecting assignments. Analysis of binding is a more tractable problem than analysis of assignments, because the semantics of binding are simpler, and there are more invariants for program-analysis tools to invoke. In particular, the invariants of the λcalculus are usually applicable to modern Lisps. {Note Difficulties with Binding}

On the other hand, the higher-order, first-class nature of Lisp functions can hinder efforts even to derive basic flow analysis information from Lisp programs. I claim it is this aspect of Lisp which is chiefly responsible for the mysterious absence of flow analysis from Lisp compilers to date. A brief discussion of traditional flow analytic techniques will show why this is so.

2   The Problem

Consider the following piece of Pascal code:

```
FOR i := 0 to 30 DO BEGIN
    s := a[i];
    IF s < 0 THEN
        a[i] := (s+4)^2
    ELSE
        a[i] := cos(s+4);
    b[i] := s+4;
    END
```

Flow analysis requires construction of a *control flow graph* for the code fragment (fig. 1).

Every vertex in the graph represents a *basic block* of code: a sequence of instructions such that the only branches into the block are branches to the beginning of the block, and the only branches from the block occur at the end of the block. The edges in the graph represent possible transfers of control between basic blocks. Having constructed the control flow graph, we can use graph algorithms to determine path invariant facts about the verteces.

In this example, for instance, we can determine that on all control paths from START to the dashed block (b[i]:=s+4; i:=i+1), the expression s+4 is evaluated with no subsequent assignments to s. Hence, by caching the result of s+4

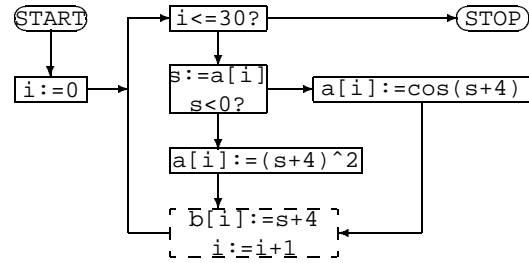
Figure 1: Control flow graph

in a temporary, we can eliminate the redundant addition in b[i]:=s+4. This information is arrived at through consideration of the paths through the control flow graph.

The problem with Lisp is that *there is no static control flow graph at compile time.* Consider the following fragment of Scheme code:

```
(let ((f (foo 7 g k))
      (h (aref a7 i j)))
  (if (< i j) (h 30) (f h)))
```

Consider the control flow of the if expression. Its graph is:
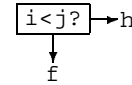
After evaluating the conditional's predicate, control can transfer either to the function that is the value of h, or to the function that is the value of f. But what's the value of f? What's the value of h? Unhappily, they are computed at run time.

If we knew all the functions that h and f could possibly be bound to, independent of program execution, we could build a control flow graph for the code fragment. So, if we wish a control flow graph for a piece of Scheme code, we need to answer the following question: for every function call in the program, what are the possible lambda expressions that call could be a jump to? But this is a flow analysis question! So with regard to flow analysis in Lisp, we are faced with the following unfortunate situation:

- In order to do flow analysis, we need a control flow graph.

- In order to determine control flow graphs, we need to do flow analysis.

Oops.

3   CPS: The Hedgehog's Representation

> The fox knows many things, but the hedgehog knows one great thing.
> – *Archilocus*

The first step towards finding a solution to this conundrum is to develop a representation for our Lisp programs suitably adapted to the task at hand. In this section, we will develop an intermediate representation language, *CPS Lisp*, which is well suited for doing flow analyis and representing Lisp programs. We will handle the full syntax of Lisp, but at one remove:

Control Flow Analysis in Scheme

Olin Shivers
Carnegie Mellon University
shivers@cs.cmu.edu

Abstract

Traditional flow analysis techniques, such as the ones typically employed by optimising Fortran compilers, do not work for Scheme-like languages. This paper presents a flow analysis technique — *control flow analysis* — which is applicable to Scheme-like languages. As a demonstration application, the information gathered by control flow analysis is used to perform a traditional flow analysis problem, induction variable elimination. Extensions and limitations are discussed.

The techniques presented in this paper are backed up by working code. They are applicable not only to Scheme, but also to related languages, such as Common Lisp and ML.

1 The Task

Flow analysis is a traditional optimising compiler technique for determining useful information about a program at compile time. Flow analysis determines path invariant facts about points in a program. A flow analysis problem is a question of the form:

> "What is true at a given point p in my program, independent of the execution path taken to p from the start of the program?"

Example domains of interest might be the following:

- Range analysis: What is the range of values that a given reference to an integer variable is constrained to lie within? Range analysis can be used, for instance, to do array bounds checking at compile time.

- Loop invariant detection: Do all possible prior assignments to a given variable reference lie outside its containing loop?

Over the last thirty years, standard techniques have been developed to answer these questions for the standard imperative, Algol-like languages (*e.g.*, Pascal, C, Ada, Bliss, and chiefly

---

Fortran). Representative texts describing these techniques are [Dragon], and in more detail, [Hecht]. Flow analysis is perhaps the chief tool in the optimising compiler writer's bag of tricks; an incomplete list of the problems that can be addressed with flow analysis includes global constant subexpression elimination, loop invariant detection, redundant assignment detection, dead code elimination, constant propagation, range analysis, code hoisting, induction variable elimination, copy propagation, live variable analysis, loop unrolling, and loop jamming.

However, these traditional flow analysis techniques have never successfully been applied to the Lisp family of computer languages. This is a curious omission. The Lisp community has had sufficient time to consider the problem. Flow analysis dates back at least to 1960, ([Dragon], pp. 516), and Lisp is one of the oldest computer programming languages currently in use, rivalled only by Fortran and COBOL.

Indeed, the Lisp community has long been concerned with the execution speed of their programs. Typical Lisp programs, such as large AI systems, are both interactive and cycle-intensive. AI researchers often find their research efforts frustrated by the necessity of waiting several hours for their enormous Lisp-based production system or back-propagation network to produce a single run. Since Lisp users are willing to pay premium prices for special computer architectures specially designed to execute Lisp rapidly, we can safely assume they are even more willing to consider compiler techniques that apply generally to target machines of any nature.

Finally, the problems addressed by flow analysis are relevant to Lisp. None of the flow analysis problems listed above are restricted to arithmetic computation; they apply just as well to symbolic processing. Furthermore, Lisp opens up new domains of interest. For example, Lisp permits weak typing and run time type checking. Type information is not statically apparent, as it is in the Algol family of languages. Type information is nonetheless extremely important to the efficient execution of Lisp programs, both to remove run time safety checks of function arguments, and to open code functions that operate on a variety of argument types. Thus flow analysis offers a tempting opportunity to perform type inference from the occurence of calls to type predicates in Lisp programs.

So it is clear that flow analysis has much potential with respect to compiling Lisp programs. Unfortunately, this potential has not been realised because Lisp is sufficiently different from the Algol family of languages that the traditional techniques