Taming the Y operator

Guillermo Juan Rozas
email: jinx@zurich.ai.mit.edu
MIT AI Laboratory, Cambridge, MA

Abstract

In this paper I present a set of conceptually simple but involved techniques used by LIAR[1], the MIT SCHEME compiler, to generate good code when recursive procedures are specified in terms of suitable versions of the Y operator. The techniques presented are general-purpose analysis and optimization tools, similar to well-known techniques used in the analysis and optimization of applicative languages, that combine synergistically to enable LIAR to generate identical machine code for ordinary recursive definitions written using `letrec` and those written using suitable forms of Y.

1 Introduction: The problem

Modern programming languages allow programmers to write local recursive procedure definitions. COMMON LISP [17] provides the `labels` special form for this purpose. SCHEME [13] provides the similar `letrec` special form, as well as internal definitions, a syntactic alternative more akin to the internal procedure declarations of ALGOL-60 [6].

Recursive procedures are described, and sometimes implemented, particularly for top-level procedures, in terms of assignment. For example, the first expression in Fig. 1 can be considered shorthand for the second.

LISP compilers typically treat `letrec` (or `labels`) as a primitive special form in order to generate good code. The expansion shown in Fig. 1 would cause typical compilers to emit code that would unnecessarily manipulate and allocate closures from the runtime heap. Compiled code might also allocate and initialize locations used to create the circular reference specified in the code (the location for `fact`, which can be viewed as the local function cell for `fact`).

[1] LIAR, and in particular, the code whose effects are described here, are the work of several people, primarily Chris Hanson and the author.

```
(letrec ((fact (lambda (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1)))))))
  fact)

(let ((fact 'anything))
  (set! fact (lambda (n)
               (if (= n 0)
                   1
                   (* n (fact (- n 1))))))
  fact)
```

Figure 1: Traditional `letrec` implementation

```
(let ((Y (lambda (f)                ; lambda-1
           ((lambda (g) (g g))      ; lambda-2
            (lambda (x)             ; lambda-3
              (f (lambda ()         ; lambda-4
                   (x x))))))))
  (Y (lambda (fg)                   ; lambda-5
       (lambda (n)                  ; lambda-6
         (if (= n 0)
             1
             (* n ((fg) (- n 1))))))))
```

Figure 2: Factorial using applicative-order Y

Treating `letrec` as a primitive form is effective, but aesthetically unsatisfying. LISP is notationally and semantically based on Church's lambda calculus [2], which does not include `letrec` or assignment. The Y combinator, written in the lambda calculus itself, can be used to express recursive procedures. For example, the expression in Fig. 2 uses an applicative-order version of the Y operator to specify a recursive procedure. This is the primary example for the rest of the paper.

Although `letrec` could be described and implemented in terms of Y, compilers rarely use this technique to implement recursive procedures.

The typical LISP compiler, when given the code for Y and recursive procedures that use Y as shown in Fig. 2, produces much worse code than when the version that uses assignment is compiled. The generated code constructs a runtime closure for each execution of a lambda expression in the code being compiled, and does not branch directly to the entry point of the code on the recursive call.

Implementing `letrec` primitively in a compiler is almost a necessity because of the inefficiencies associated with not doing so, yet it is not conceptually necessary. This paper shows how a compiler can use independently-useful general-purpose optimization techniques to effectively eliminate Y from a program.

2 Overview

LIAR has a large collection of analysis passes in the front end that guide back end code generation. In this paper, I describe informally only those analysis techniques that affect the structure of the code generated for recursive procedures expressed with Y. These techniques are similar to well-known techniques, and were added to LIAR as part of its general-purpose bag of optimization tricks. They combine profitably to eliminate certain forms of Y, but were not devised with this purpose in mind.

The techniques involved in eliminating Y from the code in Fig. 2 are the following:

- Data flow analysis allows the compiler to find the recursive call, and allows it to deduce that code pointers are not necessary when representing the closures of most of the lambda expressions that appear in the code.

- Environment optimization allows the compiler to establish that environment pointers are not necessary when representing the closures of most of the lambda expressions that appear in the code.

- Side-effect analysis allows the compiler to eliminate all of the code in Y proper because the circular reference and the return value have been established by the data flow analysis, and the remaining code in Y has no side effects.

In this paper I also examine some of the limitations of the method presented, consider how they might be overcome, and point out a somewhat surprising and controversial consequence of the techniques presented here.

3 Data flow analysis

When presented with the code in Fig. 2, it takes some observation to deduce that the expression `(fg)` always returns a closure of the lambda expression labeled `lambda-6` and thus that `((fg) (- n 1))` is a recursive call. This section describes how LIAR arrives at that conclusion.

Consider all the potential executions of a program, and the potentially infinite set of values created, manipulated, and passed around at run time. We can classify these potential values into a finite number of sets, choose a representative for each set and run the program operating on the representatives rather than on the actual values. If our sets are chosen with sufficient resolution, we may be able to prove, at compile time, properties about the values that variables will hold at run time.

In the trivial classification, a single representative is chosen for all potential values. We can call this representative *unknown*. This choice would result in the same effect as what many compilers achieve. Compilers usually handle variable bindings by generating code that allocates space to hold the value (in registers, the stack, or the heap), and treat variable references and assignment by fetching and storing into the location. After all, nothing is known at compile time about the actual values that will be stored at run time and this prevents the compiler from optimizing allocation and references.

A more interesting situation arises when we use representatives with finer resolution. Within LIAR's data flow analyzer, constants appearing in the program represent themselves, each lambda expression represents all its possible runtime closures, and an *unknown* value represents the values of top-level free variables and the results of procedures not visible at compile time, including primitives.[2]

These values can be propagated by running the code at compile time using the representatives rather than the actual code, and accumulating the representative values that arrive at each variable location until the system reaches a steady state, that is, no variable receives a representative value that had not already been accumulated for it. At the end of the propagation process, associated with each variable in the original program we will have a set of canonical values representing the actual values that may reach it at run time.

After this propagation, if one of the values accumulated is the *unknown* value, then nothing can be assumed about the value of the instances of the variable. The com-

[2]The resolution of the analysis can be improved in many ways. We can choose a canonical pair object for all the values returned by `cons`, a canonical unspecified number for all the values returned by `+`, etc. and perform some limited form of type and range checking based on the results of the propagation. LIAR does not currently do this.

```
(lambda (n)                ; outer lambda
  (let ((f (lambda (g m) ; inner lambda
             (if (= m 0)
                 1
                 (* m (g g (- m 1)))))))
    (f f n)))
```

Figure 3: Factorial using self-application

```
(lambda (n)                ; outer lambda
  (<reference to inner lambda> n))

(lambda (m)                ; inner lambda
  (if (= m 0)
      1
      (* m (<reference to inner lambda>
            (- m 1)))))
```

Figure 4: Self-applicative factorial after data-flow analysis

piler will have to represent the variable in the ordinary way. Otherwise, the compiler can assume that the intersection of the properties of the representatives will be true of the actual values manipulated at run time. For example, if all the canonical values that reach a particular variable are lambda expressions with three formal parameters, then the compiler can assume that all runtime values that will reach all instances of the variable will be procedures of exactly three arguments, and thus avoid checking the number of arguments for calls to the values of the variable.

A very important and common case occurs when only one representative reaches a given variable. We say that such a variable has its value known at compile time, meaning that the class of values to which actual runtime values belong is known at compile time. Anything that is true of all values in the class of the representative can be assumed about the values that will arrive at run time. In particular, the variable can be completely eliminated if the value is a constant. If the single value is a lambda expression the representation of the closures corresponding to the lambda expression may be optimized. For example, a procedure call whose operator is such a variable can transfer control by a direct branch instruction to the code location for the known lambda expression found at compile time - the compiler has proven that the call will only be able to invoke closures of the known lambda expression.

If the lambda expression is only propagated to variables whose value is known to be the lambda expression, no code pointer is necessary in the runtime representation of the lambda expression's closures, since all call points will directly branch to the target code. In addition, if the free variables of the lambda expression are available at all points where the closures might be called, there is no need to create runtime closures at all for the lambda expression. The environment preparation can be done at each call point, and the free variables need not be collected at the point where the lambda expression appears. The compiler can eliminate those variables whose known values are found to be lambda expressions that do not require runtime closures.

Consider the code in Fig. 3. After representative value propagation, variables `f` and `g` will have a known value, namely the inner lambda expression. Since this lambda expression has no significant free variables, i.e. its free variables are those (namely `=`, `*`, and `-`) that are free at top-level and accessible through simple direct mechanisms, there is no need to capture an environment, and the compiler can deduce that the inner lambda expression need not be closed at run time. Variables whose only value is this lambda expression (namely `f` and `g`) can be safely removed. In addition, the expression `(g g (- n 1))` has been proven to be a call to (the single closing of) the inner lambda expression, in which the `g` parameter has been dropped. The remains of the code are now as in Fig. 4.

Note the similarity between the end result of the analysis and what would have occurred if the recursive definition had been written using `letrec`.

Consider again the example in Fig. 2. After representative value propagation, the compiler will have established that `f` will have as its only values closings of `lambda-5`, `x` and `g` will only take closings of `lambda-3`, and `fg` will only have closings of `lambda-4`. The propagation will also prove that the value returned by the complete expression and the values returned by `lambda-4`, and therefore the procedures to which the values of `(- n 1)` will be given, will all be closings of `lambda-6`. Examining the significant free variables immediately removes the binding for `Y` (its value has no free variables), but it is not clear that any other variables can be removed. The remaining lambda expressions have free variables whose values are closings of lambda expressions, and we have not yet proven that the free variables of such a lambda expression are either known at compile time or available at the point of their calls. That is the subject of the next section.

4 Environment optimization by Lambda drifting

Consider the simple code in Fig. 5. Since `lambda-2` has

```
(lambda ()                          ; lambda-0
  (let ((f (lambda (x)              ; lambda-1
             (if (= x 0)
                 1
                 (+ x (* x x))))))
    (lambda (x y)                   ; lambda-2
      (* (f x) (f y)))))
```

Figure 5: Simple Environment optimization example

```
(letrec
    ((foo (lambda (x) ... bar ...))
     (bar (letrec
              ((baz (lambda (y) ... quux ...))
               (quux (lambda (z) ... foo ...
                                 ... baz ...)))
            (lambda (w)
              ... quux ...))))
  ... foo ...)
```

Figure 6: Circular environment dependencies

a free variable, a simple-minded compiler would collect the runtime values of the variable in the corresponding closures generated for `lambda-2`. However, the lexical structure of the code is used only for hiding the procedure named `f`. If `lambda-1` and `lambda-2` appeared (and were defined) at top level, a simple minded compiler would know not to close them.

LIAR's environment optimizer *drifts* such lambda expressions towards the root, to avoid closing over variables that could otherwise be available. The example in Fig. 5 is too simple, since the drifting can be accomplished by first noticing that `lambda-1` has no free variables and can therefore drift to the root. `lambda-2` can then drift to the root since the value of its only free variable is available there. LIAR's drifting algorithm handles more complicated cases - it was designed to reduce the number of closures in code with circular dependencies such as that appearing in Fig. 6.

In this example, all lambda expressions can be drifted to the root simultaneously, because all their free variables are known to have as their value some of the other lambda expressions, which have also drifted to the root, and thus the values of their free variables are available at their position in the new environment tree. Access to the variables is not necessary since the values that they hold are known. Since the variables are no longer necessary, the names are removed by another of LIAR's passes.

LIAR's drifting algorithm is based on the following two constraints:

1. A lambda expression must not drift past the frame that binds one of its free variables if the value of the variable is not known at compile time.

2. A lambda expression must not drift past the frame that binds one of its free variables whose value is known at compile time to be (a closure of) another lambda expression unless the latter is positioned at the same level or closer to the root, and therefore directly accessible.

LIAR initializes the target position for each lambda expression to be that frame farthest from the root in the environment tree that binds one of the lambda expression's free variables whose value is not known. As long as the lambda expression does not drift closer to the root, the first constraint will be satisfied. The iterative phase described below will only move them further from the root.

LIAR then constructs the graph of lambda-expression dependencies based on the second constraint, and proceeds to iteratively examine all the lambda expressions to guarantee that the second constraint is satisfied. When LIAR finds a lambda expression for which the second constraint is not satisfied, it changes the lambda expression's target position to be the frame closest to the root at which the constraint is satisfied, and uses the graph of dependencies to queue for re-examination those lambda expressions that may have been affected by the change. When the queue is empty, both constraints are satisfied for each lambda expression and the target position has been found.[3]

The algorithm always terminates because no lambda expression ever drifts deeper than its position in the source program, and the depth of the target position of each lambda expression never decreases after the initial assignment.

In the example from Fig. 2, the first constraint is moot, since there are no variables bound in inner frames whose values are not known. The initial target position for all the lambda expressions is the root environment, where the second constraint is satisfied for each lambda expression, and none is queued for re-examination.

After lambda drifting and removing now-unnecessary names because their values are known and accessible, LIAR's internal model of the code is approximately what appears in Fig. 7. Note that the residual calls to the paremeterless procedures derived from `lambda-1` through `lambda-5` would now be only executed for effect, since the value returned by them is known.

[3] A later pass of the compiler, namely the closure-format designer, occasionally undrifts some lambda expressions further to avoid generating more closures at run time. This pass of the compiler, although interesting on its own, is not relevant to the issue at hand.

```
;; top level expression:
(begin
  (<reference to lambda-1>) ; for effect
  <reference to lambda-6>)

(lambda ()                  ; lambda-1 (Y)
  (<reference to lambda-2>) ; for effect
  <reference to lambda-6>)  ; returned value known

(lambda ()                  ; lambda-2
  (<reference to lambda-3>) ; for effect
  <reference to lambda-6>)  ; returned value known

(lambda ()                  ; lambda-3
  (<reference to lambda-5>) ; for effect
  <reference to lambda-6>)  ; returned value known

(lambda ()                  ; lambda-5
  <reference to lambda-6>)  ; returned value known

(lambda (n)                 ; lambda-6 (factorial)
  (if (= n 0)
      1
      (* n ((begin
              (<reference to lambda-4>) ; for effect
              <reference to lambda-6>)
            (- n 1)))))

(lambda ()                  ; lambda-4
  (<reference to lambda-3>) ; for effect
  <reference to lambda-6>)  ; returned value known
```

Figure 7: Code after environment optimization

```
(lambda (n)                  ; top-level lambda
  (let ((accum 1))
    (letrec ((loop
              (lambda (i)    ; inner lambda
                (if (> i n)
                    accum
                    (begin
                      (set! accum (* i accum))
                      (loop (1+ i)))))))
      (loop 1))))
```

Figure 8: A side-effect free procedure

5 Side-effect analysis to eliminate useless code

The side-effect analysis performed by the front end of LIAR is straightforward. It is currently used only to eliminate calls that return known values and has no understanding of procedures that return values that may be affected by other procedures. A richer model would be necessary to allow LIAR to share the results of calls, and to re-order code when side effects do not interfere.

The LIAR front end classifies side effects into variable assignments (via `set!`), which are syntactically visible and therefore simpler to manage, and mutation of arguments or hidden state performed by primitives (e.g. `set-car!` or `write`).

Primitive procedures of the language are classified according to whether they are side-effect free or not. A primitive is considered side-effect free if it does not perform side effects itself, even though its operation may observe the side effects of other procedures (e.g. The mutation of the first component of a pair is charged to the `set-car!` operation that causes it, not to the `car` operation that notices it). The *unknown* procedure is treated as a side-effecting primitive.

The data flow analysis described above is used to compute the complete call graph, which may include the *unknown* procedure. Once a procedure, primitive or not, has been tagged as potentially side-effecting, all of its callers are contaminated transitively. After the side-effect information is propagated, user-defined procedures that have not been contaminated are assumed to be side-effect free, and calls to them can be eliminated if their values are known.

The contamination is careful to mask the effects of variable assignment to local variables. For example the top-level lambda expression in Fig. 5 is considered to be side-effect free although it performs its operation by using variable assignments internally. The inner lambda expression is not side-effect free, obviously.

```
;; top level expression
<reference to lambda-6>

(lambda (n)          ; lambda-6 (factorial)
  (if (= n 0)
      1
      (* n (<reference to lambda-6>
            (- n 1)))))

(lambda ()           ; lambda-1 (Y)
  <reference to lambda-6>)
```

Figure 9: Code after side-effect analysis

```
(let ((Y (lambda (f)
           ((lambda (g) (f (g g)))
            (lambda (g) (f (g g)))))))
  (Y (lambda (f)
       (lambda (n)
         (if (= n 0)
             1
             (* n (f (- n 1))))))))
```

Figure 10: Recursive factorial using normal-order Y

In our ongoing example, after side-effect analysis, the code would be modeled by the code in Fig. 9, with `lambda-2` through `lambda-5` identical to `lambda-1`. The compiler would then eliminate the useless lambda expressions (`lambda-1` through `lambda-5`) by only emitting code for those that are transitively reached from the top-level expression, arriving at the desired code.

6 Unexpected consequences

Given the techniques described above, the expression in Fig. 10 compiles into the same code as the expression in Fig. 2, even though without the analysis and optimization, the version that uses the normal-order version of the Y combinator would never terminate!

The difference in behavior with respect to termination is not specific to the normal-order version of Y: we run into it with simpler programs such as `cdrloop` from Fig. 11. The compiler turns the recursive call in `cdrloop` into a return of the constant `DONE`, since `cdrloop` has no side effects, and can only return `DONE`. Thus `cdrloop` immediately returns `DONE` for all arguments, including circular lists and non-lists.

```
(letrec ((cdrloop
          (lambda (x)
            (if (null? x)
                'DONE
                (cdrloop (cdr x))))))
  cdrloop)
```

Figure 11: A trivial `cdr` loop

LIAR only performs this transformation in the presence of suitable declarations that allow it to early-bind `null?` and `cdr` so that it knows that they are side-effect free and therefore that the whole lambda expression is free of side effects. In addition, the transformation is only performed if the code is compiled unsafely, that is, with type-checking in primitive operations such as `cdr` disabled. The type-checking version immediately introduces an *unknown* side-effect in the implicit call to `error`, and under these circumstances, the compiler will not consider `cdrloop` side-effect free, and the recursive call will not be eliminated.

The difference in behavior with respect to termination (and errors in the case of `cdrloop`) occurs because the compiler eliminates calls to procedures without side effects whose returned values are known, but it is not bothering to prove that the procedures would actually ever return. In other words, there is no termination analysis, and the optimization is performed without regard to maintaining this behavior. The compiler has not really proven that a particular value will be returned, but has proven instead that no other value can be returned, and takes this to mean that the value is in fact returned.

Termination analysis is, of course, undecidable in the general case (see [9], for example), and attempting to make code such as the above behave as when compiled naively (as, for example, by the compiler in [1]) would in all likelihood break the optimization for the cases in which it is not controversial.

Whether optimization under these circumstances is allowable is debatable, but defensible. Informally, code where calls have been removed in this way will behave identically to the original code for those values for which the original terminates normally (i.e. without errors). This means that ordinary code that merely uses the original version will not notice if the optimized version is substituted for the original. In the absence of error (and interrupt) handling facilities or a language requirement that all errors be detected and reported, the optimization is safe within the system, since no program will ever report *different* answers if the optimized version is substituted for the original.

Somewhat less informally, programs `P` and `Q` are equivalent if there does not exist a function Φ such that $\Phi(M[\mathtt{P}]) = 1$ while $\Phi(M[\mathtt{Q}]) = 2$, where M is the semantic function that maps programs to their functional meaning.

I do not view this definition as useful or particularly desirable when considering program optimization, since it translates into whether an external observer who possesses an oracle can distinguish between both programs. In the long run, we will want programs to design, write, and test other programs with no human intervention, and therefore not constrain what an optimizer can do to satisfy a human who will never directly interact with the components of the program.

Consider the following definition: Programs P and Q are indistinguishable *by other programs* if there is no program F such that $M[\text{F}\circ\text{P}] = 1$ while $M[\text{F}\circ\text{Q}] = 2$.

In my view, an optimizer's job is not to produce a program equivalent to its input but to produce a program indistinguishable from its input *by other programs*, which runs faster (or at least not more slowly) than the original.

Of course, a program whose meaning is the divergent computation is indistinguishable from any other program by other programs, but runs more slowly than they do, and thus no optimizer should ever produce a program that does not terminate or is in error when the input terminates normally.

This non-standard notion of program equivalence is related to the concept of *observational equivalence* (see [11]).

To satisfy those who desire more predictability from their compiler, LIAR currently provides a switch that turns off the side-effect analysis pass by marking everything as potentially side-effecting. By using it, the user can prevent LIAR from removing procedure calls that may not terminate or cause an error at run time.

7 Comments and limitations

The techniques presented above are useful as general-purpose analysis and optimization tools, and were included in LIAR independently to allow it to generate better code in general. Nevertheless, they combine constructively to effectively remove some versions of Y and of its mutual-recursion cousins from programs.

The data flow analyzer was originally written in order to perform escape analysis leading to procedure object representation. In addition, it is now used to determine environment representation, and to perform various classical optimizations such as constant folding and copy propagation. As an example of one of its other uses, examining the set of potential CPS-continuations[4] that may be passed to a procedure allows the compiler to determine whether the free variables of lambda expressions are always present at constant offsets on the stack when they are invoked, and the compiler can use this information to avoid generating code that passes environment pointers to lexically inferior contexts.

[4] CPS is ***continuation passing style***, a rewriting technique used to make control explicit in the program, and often used by SCHEME compilers. See [16] for a description of the rewriting process and its use it in a compiler.

```
(let ((Y (lambda (f)                    ; lambda-1
           ((lambda (x)                 ; lambda-3a
              (f (lambda ()             ; lambda-4a
                   (x x))))
            (lambda (x)                 ; lambda-3b
              (f (lambda ()             ; lambda-4b
                   (x x))))))))
  (Y (lambda (fg)                       ; lambda-5
       (lambda (n)                      ; lambda-6
         (if (= n 0)
             1
             (* n ((fg) (- n 1))))))))
```

Figure 12: Another applicative order version of Y

As mentioned earlier, the environment adjusting code was originally written to reduce the number of procedures represented as closures. The overall informal goal of the escape analysis and environment adjustment was that if a piece of SCHEME code could be easily transcribed into PASCAL code (requiring no closures) then the corresponding SCHEME code should be translated into code that manipulates no closures.

The techniques, however, do not work as well as would be desirable. They are particularly sensitive to the details of the code used to represent the Y operator. For example, the minor variation on our ongoing example that appears in Fig. 12 does not generate the same code. The problem arises from the introduced ambiguity in the value of `fg`. Variable `fg` may now take as values closures of `lambda-4a` and closures of `lambda-4b`. In fact, it will have a closure of `lambda-4a` as its value in the outermost level of recursion, and closures of `lambda-4b` in all inner levels. Given that the value of `fg` is not known, `lambda-6` is represented as a closure over `fg`, and this contaminates some of the other lambda expressions. The compiler still proves that the call with argument `(- n 1)` is a call to a closure of `lambda-6`, and generates code that directly branches there, but there is added overhead to create and manipulate the closure objects.

This particular case could easily be solved by a front-end redundant subexpression elimination pass that merged `lambda-3a` and `lambda-3b` before the data-flow analyzer was invoked. On the other hand, if we are only interested in replacing `letrec` by macro-expanding it into uses of Y, the writer of the macro could choose expansions that do not confuse the compiler.

8 Comparisons with other work

The data flow analysis method mentioned above is an example of abstract interpretation (see [3]) of the program at compile time. It is essentially the same as Olin Shivers's 0CFA as described in his Ph.D. dissertation [15] but they were developed independently. Our version has been in place in the LIAR compiler since late 1983 [14]. In [15] Shivers also describes a finer resolution analysis, which he calls 1CFA, adding some finite trace information to the propagated representative values that allows him to resolve cases where the simpler analysis described here fails. He also discusses how to use the results of the analysis to implement several optimizations. LIAR implements variants of many of these using the simpler analysis, which appears sufficient in practice. With respect to eliminating the Y operator from code, his analysis should work at least as well as ours, and may be able to eliminate other forms without the need for source common-subexpression elimination or similar tricks.

The environment optimization technique described is only one of several present in LIAR and is also used to reduce the number of static and dynamic links [7]. The most closely related technique is lambda lifting [12]. Both techniques migrate lambda expressions toward the root of the environment tree in order to simplify environment structure. When lambda lifting, parameter lists of migrated procedures are extended to include their free variables. Calls to migrated procedures must pass their values in addition to the original arguments. Lambda drifting, the technique described above, does not extend parameter lists, since lambda expressions are only moved outside of enclosing lambda expressions if the free variables no longer accessible have known and accessible values. In essence, lambda lifting is a technique for flattening environment structure and making variable reference more regular without removing any variables. Lambda drifting is a technique that simplifies environment structure by moving lambda expressions to places where they can be accessed directly by others and thus eliminating the need for variables to hold their closures. Both techniques can be profitably used, using lambda drifting to reduce the number of free variables, and then lambda lifting to collect the remaining variables. LIAR implicitly performs lambda lifting when designing the format of procedure objects after using lambda drifting.

The side-effect analysis in the front end of LIAR is straightforward and was added with Y in mind. Other SCHEME compilers [16] have used more complex analysis to reorder expressions at the source level. The FX language [10, 8], a derivative of SCHEME, has been designed with effect analysis and effect masking at its core, and its syntax includes effect declarations that are propagated and checked statically by the compiler much in the same way that types are propagated and checked. Recent versions of the language include a type and effect reconstructor that reduces the need for declarations. With respect to eliminating Y, the simple side-effect analysis described appears to suffice.

The tension in LIAR between aggressive optimization and conformance with low-level models of execution is present in other compiler and optimization work. For example, [5] describes an aggressive optimizer based on partial evaluation which does not preserve the termination properties of the input program. This tension is also present in modern computer architecture. See, for example, [4] for a description of an architecture with imprecise traps.

9 Conclusion

In this paper I have presented a set of simple techniques that allow LIAR to effectively eliminate uses of the Y combinator. Although the techniques presented are not perfect, they show that it is feasible to define recursive procedures purely in terms of suitable versions of Y, with no performance penalty, and thus compilers need not implement `letrec` or `labels` primitively. The techniques involved are general-purpose optimization techniques, desirable for their applicability to general code, but they combine constructively to effectively remove Y.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Principles of Programming Languages*, Los Angeles, California, 1977.

[4] Digital Equipment Corp. *Alpha Architecture Reference*, March 1992.

[5] D. Weise et al. Automatic online partial evaluation. In *Proc. of the Conference on Functional Programming Languages and Architectures.* Springer Verlag, 1991.

[6] Peter Naur et al. Revised report on the algorithmic language Algol-60. *Communications of the ACM*, 6(1):1–17, January 1963.

[7] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[8] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th Principles of Programming Languages*, 1991.

[9] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[10] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. 15th Principles of Programming Languages*, 1988.

[11] Ketan Mulmuley. *Full abstraction and semantic equivalence.* MIT Press, 1986. ACM Doctoral Dissertation Award, 1986.

[12] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[13] Jonathan Rees and William Clinger (*editors*). Revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21(12), December 1986. Also available as MIT AI Memo 818a.

[14] Guillermo J. Rozas. Liar, an Algol-like compiler for Scheme. S.B. thesis, Mass. Inst. of Technology, Dept. of Electrical Engineering and Computer Science, January 1984.

[15] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda.* Ph.D. thesis, Carnegie Mellon University, May 1991. Available as CMU-CS-91-145.

[16] Guy Lewis Steele Jr. Rabbit: A compiler for Scheme. S.M. thesis, Mass. Inst. of Technology, 1978. Also available as MIT AI Memo 474.

[17] Guy Lewis Steele Jr. *Common LISP The Language, 2nd Edition.* Digital Press, 1990.