Eager Evaluation Isn’t Eager Enough
A Transformation Based Approach to Semantics-Directed Code Generation

Arthur Nunes-Harwitt
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, New York 14623
anhn@cs.rit.edu

ABSTRACT
An interpreter is a concise definition of the semantics of a programming language and is easily implemented. A compiler is more difficult to construct, but the code that it generates runs faster than interpreted code. This paper introduces rules for staging an interpreter so that it generates a compiler. An extended example suggests the utility of the technique. The rules are described formally and correctness is discussed. Finally, this technique is compared to staging and partial evaluation.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—Compilers; D.2.2 [Software Engineering]: Miscellaneous—Rapid Prototyping

General Terms
Verification, Performance

Keywords
meta-programming, compilers, interpreters, partial evaluation, staging

1. INTRODUCTION
A semantics-directed code generator is a code generator that has been derived from a semantic specification such as an interpreter. Semantics-based approaches to code generation have a number of benefits: correctness, ease of implementation, maintainability, and rational justification. Two common techniques for semantics-based code generation are partial evaluation and staging.

Partial evaluation[11] is a transformation technique for specializing programs. Program specialization can mean simply replacing some of a function’s parameters with values; however, specialization is usually understood to involve Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IJC ’14 August 14 – 17 2014, Montreal, QC, Canada
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2931-6/14/08$15.00.
http://dx.doi.org/10.1145/2635648.2635652

ting time improvements to achieve good specialization.

A staged computation is a computation that is organized so that part of the computation occurs at one stage, or time, and the rest of the computation occurs at another. Partial evaluation is a technique for staging, but this notion has broader scope. For example, it includes manual techniques such as Marc Feeley’s closure based approach to code generation[7] and related techniques that generate text. Although William L. Scherlis developed a form of equational reasoning similar to Bustall and Darlington, subsequently he and Ulrik Jørring[12] identified staging as a way to produce a code generator. No rules for staging have been established. Instead the emphasis more recently has been on creating type systems for statically typed programming languages with quotation[15, 14, 17].

This paper is concerned with an alternative approach to semantics-directed code generation that lies between staging and partial evaluation. It makes the following contributions. It identifies a new technique for staging an interpreter and deriving a code generator in the form of four essential transformations. The technique provides more guidance than traditional staging and can be used to derive a compiler, but is not fully automated like partial evaluation. Motivation is provided for this technique. An extended example suggests the utility of the technique. The transformations are presented formally, and correctness is discussed. Finally, this technique is compared to staging and partial evaluation.

2. TRANSFORMATIONS
The motivation for this transformation technique comes
from denotational-semantics\[16, 20\]. A denotational definition can be understood as an interpreter. It is less commonly recognized that the denotational definition can also be understood as a compiler: given a term, we are free to evaluate the recursive calls and derive a $\lambda$-term. Yet the eager evaluation strategy prevents reducing the applications inside abstractions. The rules described below concern answering the question: How can a denotational-style interpreter be modified so that it too generates a $\lambda$-term when using an eager evaluation strategy?

2.1 Currying Dynamic Variables

Currying is a mathematical trick to make all functions take one argument; it transforms a function of two arguments into a function of one argument that returns a function. For example, the multiplication function $m(x, y) = x \times y$ becomes $m(x) = \lambda y.x \times y$. If we have in mind that $x$ is known statically, but $y$ is known dynamically, then applying the curried form to a statically known value specializes the multiplication function. For example, applying $m$ to 2 results in the following term: $m(2) = \lambda y.2 \times y$. Thus the application of a curried function is a weak form of code generation.

Many programming languages, especially today, allow for first-class functions. In Scheme[13], the multiplication example looks as follows.

```
(define (m x y) (* x y))
```

When curried, it becomes the following.

```
(define (m x) (lambda (y) (* ,x y)))
```

However, applying $m$ to 2 yields an opaque result rather than the desired term. Something more is needed to see the text of the resulting procedure.

2.2 Code Via Quoting

To fix the problem in section 2.1, we want to see the text of the function rather than the function itself (which may not be displayable). To return text, rather than a function, we can use Scheme’s quotation and un-quotation mechanisms: backquote and comma. Upon making this change, the term comes out as expected, although now `eval` is needed to actually apply this function.

```
(define (m x) '((lambda (y) (* ,x y)))
```

But consider the following more complicated example of raising $b$ to the $n$th power and what happens when applying these currying and quoting transformations.

```
(define (p n b) ; original
  (if (= n 0)
    1
    (* b ((p (- n 1)) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b)))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b)))))
```

The result this time is inadequate because a substantial amount of static computation remains. In particular, the conditional does not depend on the parameter $b$ and should not be there. The code generated also assumes a run-time environment in which the curried form of $p$ is defined. Of course, the goal is to eliminate the need for such a run-time function.

2.3 Lambda Lowering

To fix the problem in section 2.2, we need to evaluate the test in the conditional. A way to do that is to move the function with the formal parameter $b$ inside the conditional after currying. Upon making this sequence of transformations, applying the code generating function does yield a simpler term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b ((p (- n 1)) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b)))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b))))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function $p$.

2.4 Expression Lifting

To fix the problem in section 2.3, we need to evaluate the recursive call. Since it resides in a $\lambda$-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b (p (- n 1) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b))))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function $p$. 

2.5 Lambda Lowering

To fix the problem in section 2.4, we need to evaluate the recursive call. Since it resides in a $\lambda$-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b (p (- n 1) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b))))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function $p$. 

2.6 Expression Lifting

To fix the problem in section 2.5, we need to evaluate the recursive call. Since it resides in a $\lambda$-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b (p (- n 1) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b))))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function $p$. 

2.7 Lambda Lowering

To fix the problem in section 2.6, we need to evaluate the recursive call. Since it resides in a $\lambda$-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b (p (- n 1) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b))
```

While the result here is better, it is still inadequate because we have not yet eliminated the reference to the function $p$. 

2.8 Expression Lifting

To fix the problem in section 2.7, we need to evaluate the recursive call. Since it resides in a $\lambda$-expression, the only way to evaluate the expression is to lift it out. Upon making this sequence of transformations, applying the code generating function yields an ungainly but fully simplified term.

```
(define (p n) ; original
  (if (= n 0)
    1
    (* b (p (- n 1) b))))
```

```
(define (p n) ; curried
  (lambda (b)
    (if (= n 0)
      1
      (* b ((p (- n 1)) b))))
```

```
(define (p n) ; quoted
  (if (= n 0)
    '1
    '(* b ((p (- n 1)) b)))))
```

> (p 3)

```
(lambda (b) (* b ((lambda (b) b) b)))))
```

The result this time is inadequate because a substantial amount of static computation remains. In particular, the conditional does not depend on the parameter $b$ and should not be there. The code generated also assumes a run-time environment in which the curried form of $p$ is defined. Of course, the goal is to eliminate the need for such a run-time function.
Although ideally the generated code would be more readable, we can make it more pleasant looking by post-processing.  
\( \lambda b \cdot (\lambda x. (\lambda y. y b)) b) \)  

2.5 Rule Ordering

The rules are performed in the following order. When applying the rules above, first the function is curried. Then the lambda lowering and expression lifting rules are applied repeatedly until those rules can no longer be applied. Finally the quoting rule is applied to all abstractions derived from the curried function.

2.6 Beyond Denotational Interpreters

These transformations will work on a denotational-style interpreter, but what about other kinds of interpreters? Indeed, for non-denotational interpreters these transformations are often insufficient. For example, consider an operational-style interpreter. In that case, it is quite reasonable to say that, just as a number evaluates to a number, an abstraction evaluates to an abstraction (or a closure):  
\( \epsilon (\lambda x. E, env) = \text{closure}(x, \lambda \text{env}. \epsilon(E, \text{env}'), \text{env}) \).  

Another way operational style interpreters can be different from denotational ones is that recursion in an operational style interpreter need not be on a smaller term. For example, iteration constructs are often defined in terms of themselves. The transformation technique will lead to infinite loops on this sort of expansive recursion. Following Gunter[9], we repair this problem in the interpreter by explicitly identifying the fixed-point and eliminating the expansive recursion.

For example, consider the following interpreter snippet for a while-loop construct. If the test expression \( B \) evaluates to \( False \) then the command \( C \) is not executed. If the test expression \( B \) evaluates to \( True \) then the command \( C \) is executed at least once. Then iteration is achieved by invoking the interpreter on the entire while command.

\[ \mathcal{I} (\text{while } B \text{ do } C, s) = \begin{cases} \mathcal{I} (B, s) = False & \text{then } s \\ \mathcal{I} (\text{while } B \text{ do } C, \mathcal{I} (C, s)) & \text{else} \end{cases} \]

After currying and lambda-lowering that snippet becomes the following:

\[ I (\text{while } B \text{ do } C) = \lambda s. \text{if } \epsilon(B)(s) = False \text{ then } s \text{ else } I (\text{while } B \text{ do } C)(I(C)(s)) \]

But the expression-lifting rule does not apply because attempting to lift \( I (\text{while } B \text{ do } C) \) will lead to non-termination. However, if we let \( g = I (\text{while } B \text{ do } C) \) it becomes apparent that this function can be computed; \( g \) is the fixed-point function.

\[ g(s) = \text{if } \epsilon(B)(s) = False \text{ then } s \text{ else } g(I(C)(s)) \]

3. EXTENDED EXAMPLE

To illustrate this technique, consider the application of regular expression matching. A regular expression matching interpreter takes a regular expression and a string, and determines if the string is in the language denoted by the regular expression. Often, the regular expression is fixed, and we would like the code that answers whether a string is in the language denoted by that fixed regular expression.

**Definition 1.** A regular expression is one of the following, where the predicate testing each option is in parentheses.

- The empty string. \( \text{null?} \)
- A character in the alphabet. \( \text{char?} \)
- The union of two regular expressions. \( \text{or?} \)
- The concatenation of two regular expressions. \( \text{cat?} \)
- The Kleene star of a regular expressions. \( \text{star?} \)

The matching algorithm is expressed in Scheme using continuation passing style; the continuation \( k \) is the property that must be satisfied by the remainder of the string. In the code below, a string is represented as a list of characters \( cl \).

\[
\begin{align*}
\text{(define } \text{match } \text{regexp } \text{cl } \text{k}) &= \text{cond }\begin{cases} \text{null? } \text{regexp} & (k \text{ cl}) \\
\text{char? } \text{regexp} & (\text{if } \text{null? } \text{cl} \text{ } \\
\text{#f} & (\text{and } \text{(eq? } \text{(car } \text{cl}) \text{ regexp} \text{ k cl)}) \\
\text{or? } \text{regexp} & (\text{or } \text{(match } \text{exp1<or } \text{regexp} \text{ cl } \text{k}) \text{ cl} \text{ k }) \\
\text{cat? } \text{regexp} & (\text{match } \text{exp1<cat } \text{regexp} \text{ cl} \text{ cl} \text{ c12}) \\
\text{star? } \text{regexp} & (\text{let } \text{loop } \text{((c12 cl1) } \text{ (match } \text{exp1<cat } \text{regexp} \text{ cl )})) \\
\text{#f} & (\text{and } \text{(eq? } \text{(car cl2) regexp} \text{ k cl2})) \\
\text{else} & (\text{error 'match 'match's input is bad'})))
\end{cases}
\end{align*}
\]

When \text{regexp} is the empty string, \text{match} invokes the continuation on the character list. Note that the initial continuation verifies that the character list is empty. When \text{regexp} is a character, \text{match} checks that the first character
in the character list is that character, and invokes the continuation on the tail of the character list. When regexp is a union, match merely tries both possibilities. When regexp is a concatenation, match recursively matches the first component and adds a check for the second to the continuation. When regexp is a Kleene star, match loops checking if either the continuation is satisfied or the pattern to be repeated is matched.

In the following sections, we will now apply the technique to this interpreter and derive a code generator.

3.1 Currying

A compiler for regular expressions must be a function that takes a regular expression; hence the dynamic parameters are cl and k. They are removed from the top-level parameter list and put into the parameter list of the λ-expression. The recursive calls are modified to account for this new protocol.

```
(define (match1 regexp)
  (lambda (cl k)
    (cond ((null? regexp) (k cl))
          ((char? regexp)
           (if (null? cl)
               #f
               (and (eq? (car cl) regexp) (k (cdr cl)))))
          ((or? regexp)
           (or ((match1 (exp1<-or regexp)) cl k)
                ((match1 (exp2<-or regexp)) cl k)))
          ((cat? regexp)
           ((match1 (exp1<-cat regexp)) cl
            (lambda (cl2)
              ((match1 (exp2<-cat regexp)) cl2 k))))
          ((star? regexp)
           (let loop ((cl2 cl))
             (or (k cl2)
                 ((match1 (exp<-star regexp)) cl2
                  (lambda (cl3)
                    (if (eq? cl2 cl3) #f (loop cl3)))))))
          (else (error 'match1 "match1's input is bad")))))
```

3.2 Lambda lowering

Since (cond (c1 c2) ... ) ≡ (if c1 c2 (cond ... )), it is possible to apply the conditional form of the lambda lowering rule several times. The lambda just below the definition in match1 is lowered into each branch of the cond-expression3.

```
(define (match2 regexp)
  (cond ((null? regexp) (lambda (cl k) (k cl)))
        ((char? regexp)
         (lambda (cl k)
           (if (null? cl)
               #f
               (and (eq? (car cl) ,regexp) (k (cdr cl))))))
        ((or? regexp)
         (let ((f1 (match2 (exp1<-or regexp)))
               (f2 (match2 (exp2<-or regexp))))
           (lambda (cl k) (or (f1 cl k) (f2 cl k)))))
        ((cat? regexp)
         (let ((f1 (match2 (exp1<-cat regexp)))
               (f2 (match2 (exp2<-cat regexp))))
           (lambda (cl k)
             (f1 cl (lambda (cl2) (f2 cl2))))))
        ((star? regexp)
         (let ((f (match2 (exp<-star regexp))))
           (lambda (cl k)
             (let loop ((cl2 cl))
               (or (k cl2)
                   (f cl2 (lambda (cl3)
                     (if (eq? cl2 cl3) #f (loop cl3))))))))
          (else (error 'match2 "match2's input is bad")))))
```

3.3 Expression lifting

Since the recursive calls have been curried and do not depend on the dynamic variables, it is possible to lift them out of the lowered lambdas. In this example, it is clear that the calls will halt since the recursive calls are always on smaller structures.

```
(define (match3 regexp)
  (cond ((null? regexp) (lambda (cl k) (k cl)))
        ((char? regexp)
         (lambda (cl k)
           (if (null? cl)
               #f
               (and (eq? (car cl) ,regexp) (k (cdr cl))))))
        ((or? regexp)
         (let ((f1 (match3 (exp1<-or regexp)))
               (f2 (match3 (exp2<-or regexp))))
           (lambda (cl k) (or (f1 cl k) (f2 cl k)))))
        ((cat? regexp)
         (let ((f1 (match3 (exp1<-cat regexp)))
               (f2 (match3 (exp2<-cat regexp))))
           (lambda (cl k)
             (f1 cl (lambda (cl2) (f2 cl2))))))
        ((star? regexp)
         (let ((f (match3 (exp<-star regexp))))
           (lambda (cl k)
             (let loop ((cl2 cl))
               (or (k cl2)
                   (f cl2 (lambda (cl3)
                     (if (eq? cl2 cl3) #f (loop cl3))))))))
          (else (error 'match3 "match3's input is bad")))))
```

3.4 Quoting

Now each λ-expression is quoted. The Scheme backquote syntax is used to allow some sub-expressions to be evaluated. In particular, non-global free variables are quoted in the text4.

```
(define (match4 regexp)
  (cond ((null? regexp) ' (lambda (cl k) (k cl)))
        ((char? regexp)
         ' (lambda (cl k)
            (if (null? cl)
                #f
                (and (eq? (car cl) ,regexp) (k (cdr cl))))))
        ((or? regexp)
         (let ((f1 (match4 (exp1<-or regexp)))
               (f2 (match4 (exp2<-or regexp))))
           ' (lambda (cl k) (or (f1 cl k) (f2 cl k)))))
        ((cat? regexp)
         (let ((f1 (match4 (exp1<-cat regexp)))
               (f2 (match4 (exp2<-cat regexp))))
           (lambda (cl k)
             (f1 cl (lambda (cl2) (f2 cl2))))))
        ((star? regexp)
         (let ((f (match4 (exp<-star regexp))))
           (lambda (cl k)
             (let loop ((cl2 cl))
               (or (k cl2)
                   (f cl2 (lambda (cl3)
                     (if (eq? cl2 cl3) #f (loop cl3))))))))
          (else (error 'match3 "match3's input is bad")))))
```

3 An exception to the rule is made in the error case; the lambda is not lowered. The motivation is practical: it is preferable to find out right away that the input is invalid.

4The formal rule for quoting in section 4 requires that the variables that are unquoted are all in a surrounding let binding. This restriction forces more locality without a loss of expressiveness; however, in practice we feel free to unquote other free variables. In the example, regexp is unquoted in the character case.
3.5 Output
When the regular expression is \(a^*\), the simplified output becomes the following.

\[
\begin{align*}
\lambda (cl\ k) \\
& \begin{cases}
   & (or (if (null? cl2) #\a)
   & (k (cdr cl2))) \\
   & (if (null? cl2)
   & #f)
   \end{cases} \\
& (let loop (((c1 cl2))
   (if (null? cl)
   #f)
   \begin{cases}
   & (and (eq? (car cl2) #\b)
   & (k (cdr cl2))))
   \end{cases})
\end{align*}
\]

3.6 A Non-Denotational Variation
Suppose we had naively defined the Kleene star operation non-denotationally in the expression interpreter. We start by modifying the interpreter in the following fashion that makes staging and partial evaluation more challenging.

\[
\begin{align*}
\lambda (cl\ k) \\
& \begin{cases}
   & (or (if (null? cl2) #\a)
   & (k (cdr cl2))) \\
   & (if (null? cl2)
   #f)
   \end{cases} \\
& (let loop (((c1 cl2))
   (if (null? cl)
   #f)
   \begin{cases}
   & (and (eq? (car cl2) #\b)
   & (k (cdr cl2))))
   \end{cases})
\end{align*}
\]

Let \(f_2 = (\text{match} \ \text{regexp})\), then \((\text{match} \ \text{regexp}) = \lambda (cl, k) \ldots (\text{match} \ \text{regexp})\) becomes \(f_2 = \lambda (cl, k) \ldots f_2 \ldots\), at which point we consider the fixed point solution of \(f_2\). To implement that idea, the code is modified as follows.

\[
\begin{align*}
\lambda (cl\ k) \\
& \begin{cases}
   & (or (if (null? cl2) #\a)
   & (k (cdr cl2))) \\
   & (if (null? cl2)
   #f)
   \end{cases} \\
& (let loop (((c1 cl2))
   (if (null? cl)
   #f)
   \begin{cases}
   & (and (eq? (car cl2) #\b)
   & (k (cdr cl2))))
   \end{cases})
\end{align*}
\]

Now the quoting rule can be applied. When performed, we get a code generator for regular expressions that includes Kleene star forms even though the interpreter was not written in a denotational-style.

4. SUMMARY AND FORMALIZATION
In this section, we formalize the interpreter language so that the transformation rules can be stated more formally. This formality allows for a discussion of correctness and other properties.

4.1 Modeling Scheme
The language the interpreter is written in is assumed to be Scheme-like. We model Scheme via a call-by-value \(\lambda\)-calculus with constants, conditionals, and quotation (see figure 1). A new kind of variable, the comma variable, is used to model Scheme's unquote. A \(\text{let}\) not involving a comma variable is understood in the usual way to abbreviate the application of an abstraction. The \(\text{eval}\) operator can be defined in terms of the \(\text{let}\) with comma variables. The semantics is similar to the \(\lambda\)-calculus with quotation found in [15].

In Scheme, both programs and data are parenthesized expressions. Data are distinguished from programs by putting a quotation mark in front. Thus \'(+ 2 3)\ performs addition, but \'(+(2 3))\ is a list. In the interpreter language above, \'(+ 2 3)\ performs addition and \'([+ 2 3])\ is data. The notation here differs somewhat from Scheme in that Scheme allows an arbitrary form to be quoted; thus \'(1 2 3)\ is simply a list of numbers. While it is technically possible to write \([1(2, 3)]\), the syntax of application exists and so the expression does not make sense. Scheme also makes it possible to plug values into a quoted form. The comma operator is used to unquote an expression. Thus the Scheme expression \(\text{let} \ldots \ldots \text{match}\)
unquoted thereby implementing the eval

let, y = e in e'

if e₀ then e₁ else e₂

Values v ::= e

Values v ::= e

let x = e in e' is syntactic sugar.

(eval e) is syntactic sugar.

Figure 1: Interpreter language syntax

((y 2)) ‘(+ , y 3)) evaluates to a list whose second component is 2. In the interpreter language above, the comma is not an operator; rather a second kind of variable is introduced, the comma-variable, which is intended to resemble Scheme’s application of the comma operator to a variable. The let-form for comma-variables is used to plug into a quoted expression. In the interpreter language, the comma example is written let , y = 2 in [+(,(y),3)]. Further, it is natural to use this let-form to define the operator eval.

The meaning of this interpreter language λ-calculus is mostly standard (see appendix A). In Scheme, we have that (let ((y ‘(+ 3 4))) ‘(* 2 ,y)) evaluates to the list (* 2 (+ 3 4)). This can be understood as removing the quotation and replacing the comma-variable with the unquoted term. In Scheme, the body of the let-form cannot merely be a comma-variable; in Scheme the comma operator must appear inside a quasi-quote form. However, in the interpreter language it is possible. Observe that when the body of the let-form is merely the comma-variable, the quoted term is unquoted thereby implementing the eval operator.

Reduction can be extended to an equivalence relation by making sure the relation is reflexive, symmetric, and transitive (see appendix B). In addition, when making arguments it is useful to be able to say that sub-structural equality implies equality. Hence those rules are added.

4.2 Formal Transformation Rules

The examples in sections 2 and 3 illustrate how a procedure can be modified so that it generates a λ-term. The key idea is the following: An expression within an abstraction cannot be evaluated, and so the code is restructured so that the expression is no longer within the abstraction.

The transformations are in figure 2. Rules (1) and (2) are about currying. The equivalence of functions and their curried counterparts is well known. Although the rules are expressed as local changes, rule (2) must be applied completely using non-local assumptions and information.

Rules (3) and (4) are about lambda lifting\(^5\). These rules involve moving an expression that is just inside an abstraction and does not depend on the parameters of an abstraction out of the abstraction. In particular, if the abstraction body is a conditional, but the conditional does not depend on the abstraction’s parameters, we may regard the conditional as specifying one of two abstractions. Or, if the abstraction body defines an intermediate value that does not depend on the parameters, we may regard the definition as occurring outside the body of the abstraction.

For rule (3), concerning a conditional, if e reduces to a value v, then the body of the abstraction depends on v. When false, the body is e₂; otherwise the body is e₁. And that is what the right-hand-side says. For rule (4), concerning a let-binding, if e reduces to a value v, then the let on the left-hand-side substitutes v for z in e₀. The let on the right-hand-side substitutes v for z in the abstraction, but it passes right through and becomes a substitution in e₀ since z is distinct from the formal parameters.

Rule (5) is expression lifting\(^6\). This rule is similar to lambda lowering insofar as both involve moving an expression out of an abstraction. However, with expression lifting, the entire expression is moved completely out of the abstraction if it does not depend on the parameters of the abstraction. Typically, the expression being lifted is an application. If e reduces to a value v, then the body of the abstraction on the left-hand-side will replace u with v. The let on the right-hand-side also ultimately replaces u with v since the substitution for z passes right through the abstraction.

\(^5\)Lambda lowering sounds similar to ‘lambda dropping’. Lambda dropping is very different involving, among other things, moving an entire abstraction rather than the lambda.

\(^6\)Extracting an expression is reminiscent of how the ANF transform works. Here the expression is moved out of an abstraction; with the ANF transform the expression is moved out of an evaluation context.
4.3 Correctness

The correctness of rules (3), (4), and (5) relies only on local reasoning (see appendix C). Note that they all assume that the evaluation of $e$ terminates. If that is not the case, looping outside of an abstraction is always observed, but looping inside an abstraction is observed only if the abstraction is called. In practice, it is clear for rules (3) and (4) whether or not $e$ terminates: typically it is a call to a structure predicate and it does not loop. The termination of $e$ in rule (5) is more subtle. If it is a recursive call on substructure it will terminate. If it is a recursive call on the same structure it will not terminate. Otherwise, termination is not obvious.

Rule (6) is about quotation. It transforms an expression that returns an abstraction into an expression that returns the text that represents that abstraction. With this rule, the transformed expression reduces to a different value from the original, and so here the notion of correctness is different. Correctness means that applying the eval operator to the text that results from reducing the transformed expression results in the same value as the original expression. This rule also requires non-local information and assumptions; it must be applied to all branches of a conditional. Intuitively, we argue that since the text looks like the expression we would have evaluated right away, and name capture is avoided, then it must be that the same value is computed. (See appendix D for a global correctness result involving an abstractly specified interpreter.)

4.4 Additional Properties

Here we briefly mention two properties concerning whether the set of rules in figure 2 is the right size.

One interesting property is that it is possible to use only rule (5) to achieve a transformation very similar to rule (4). Nevertheless, it is convenient to use rule (4). Further rule (4) avoids the introduction of an extra variable; an additional rule would be needed to eliminate the extra variable.

Another interesting property concerns whether there are enough rules. For the interpreter language in figure 1, we argue informally that no additional rules are necessary. The currying rules introduce an abstraction from which sub-expressions can be extracted, the quoting rule turns that abstraction into text, and the remaining rules extract sub-expressions from that abstraction. Assuming the body of the abstraction has no sub-expressions involving quotation, there are only five cases. For variables and constants, there is nothing to extract. If the body is also an abstraction, we assume the rules are enough for that abstraction, and use them again. If the body is an application, use rule (5) on either the sub-expressions or the entire expression; that is all that can be done. If the body is a conditional, use rule (5) on sub-expressions or use rule (3) if only the first sub-expression is independent of the parameters; that is all that can be done.

5. COMPARISON OF TECHNIQUES

The transformation technique presented in this paper is a manual technique. Another manual technique is staging. The work in staging assumes the programmer guesses a staged form of an algorithm, and then provides a type-checking algorithm that verifies the staging has been done correctly. The transformation technique here is complementary since it helps the programmer perform the staging.

While the ideas underlying partial evaluation can often be used effectively to manually derive a sophisticated algorithm from a naive one, that is not the case when attempting to derive a code generator. Manually partially evaluating an interpreter on a particular input may yield code for that input, but deriving a code generator traditionally requires at least the second Futamura projection[8]: applying the partial evaluator to itself. Manually partially evaluating the partial evaluator with an interpreter is unwieldy. In contrast, the transformation technique in this paper can often be used to manually derive a code generator from an interpreter.

The cogen approach[1, 18] is an alternative to traditional partial evaluation. Like the technique presented here, the emphasis is on generating a code generator. The cogen approach borrows from the ideas involved in off-line partial evaluation. To create a code generator, a binding time analysis is performed and the input program is annotated. Instead of using the annotated program for partial evaluation, the annotations are reified to generate the generator. Then the generator can be used for partial evaluation, if desired. However, if a derivation is desired, a separate binding time analysis is less direct than the transformation technique discussed in this paper.

Partial evaluators are fully automatic. This may make partial evaluation more attractive for some applications; yet it seems possible to at least partially automate the application of the transformations. Implementing currying appears straightforward, but would involve a control flow analysis. Depending on the level of automation desired, one difficulty when implementing rules (3), (4), and (5) involves verifying that particular terms terminate. Another difficulty is if some parameters, such as the store, are implicit. Finally, quoting may be the trickiest to implement, because one would like to relax the restriction that the free variables are all locally let-bound, and the semantics of Scheme unquoting is more complicated than the model of unquoting in this paper.

There are interpreters for which the transformation technique does not succeed. That is also the case for partial evaluation algorithms. Early partial evaluators had trouble with assignment and/or higher-order functions[4, 3]. Coming up with the right binding time improvements to help a partial evaluator can be challenging because partial evaluation algorithms are quite complicated[6]. In contrast, because the individual transformations that form the transformation technique are so simple it is easier to identify the necessary changes in an interpreter so that a compiler can be generated than guessing binding time improvements.

6. CONCLUSION

This paper has presented a new transformation technique for deriving a code generator from an interpreter. The transformations were presented formally, and proved correct. We have provided an example that illustrates the ideas. Finally, we argue that this technique is a worthwhile alternative to partial evaluation and staging.

A number of questions remain that deserve investigation. For example, while the transformation techniques from section 4 can be effectively applied to any denotational-style interpreter, it is not yet clear to what extent that class of interpreters can be extended. We anticipate investigating whether it is possible to formulate rules that transform an
operational semantics into a denotational one.

The longest example considered in this paper is still fairly short. A practical test for this technique would involve applying it to large examples. We have been experimenting with Prolog implementations of intermediate size and anticipate reporting on the results in a another paper.

Finally, manual transformation is a double-edged sword. It is more flexible than automatic transformation, yet it allows for the introduction of human error. It may be worthwhile to create software tools to help perform some of the suggested transformations.

7. ACKNOWLEDGEMENTS

I would like to thank Axel Schreiner and Melissa Nunes-Harwitt for carefully reading previous drafts of this paper. I would like to thank Matthew Fluet for reading a draft and for looking carefully at my proofs. I would also like to thank the anonymous referees for their valuable comments.

8. REFERENCES


APPENDIX

A. OPERATIONAL SEMANTICS

Definition 2. The form let x = e in e′ is syntactic sugar for (λx.e′)(e).

Definition 3. The form eval(e) is syntactic sugar for let , y = e in , y.

\[ \begin{align*}
\delta(c_{op}, v) &= v' \\
\delta(c_{op}(v)) &= v'
\end{align*} \]

\[ \begin{align*}
(λx.e)(v) &= e[x := v] \\
\text{if } v \neq [c] & \text{ then } e_1 \text{ else } e_2 \rightarrow e_1 \\
\text{if } v \neq \text{False} & \text{ then } e_1 \text{ else } e_2 \rightarrow e_2
\end{align*} \]

B. TERM EQUALITY

\[ \begin{align*}
e &= e \quad \text{reflexive} \\
e &= e' \quad \text{symmetric} \\
e' &= e \quad \text{transitive}
\end{align*} \]
C. LOCAL CORRECTNESS THEOREMS

Note that equality below is term equality.

Lemma 1. If \( e \rightarrow^* v \) then (if \( e \) then \( e_1 \) else \( e_2 \)) \( \rightarrow^* \)

Theorem 1. If \( e \rightarrow^* v \), and \( x_i \notin \text{FV}(e) \) then \( \lambda x. \text{if } e \text{ then } e_1 \text{ else } e_2 \), if \( e \) then \( \lambda x. e_1 \) else \( \lambda x. e_2 \).

Proof. By case analysis on \( v \).

- Suppose \( v \neq \text{False} \).
  \[ \lambda x. \text{if } e \text{ then } e_1 \text{ else } e_2 = \lambda x. \text{if } v \text{ then } e_1 \text{ else } e_2 = \lambda x. e_1 \]
  \[ = \text{if } v \text{ then } \lambda x. e_1 \text{ else } \lambda x. e_2 \]
  \[ = \text{if } e \text{ then } \lambda x. e_1 \text{ else } \lambda x. e_2 \]

- Suppose \( v = \text{False} \).
  The argument is similar.

Lemma 2. If \( e \rightarrow^* v \) then (let \( z = e \) in \( e_b \)) \( \rightarrow^* \) (let \( z = v \) in \( e_b \)).

Theorem 2. If \( e \rightarrow^* v \), \( z \neq x_i \), and \( x_i \notin \text{FV}(e) \) then let \( z = e \) in \( \lambda x. e_b = \lambda x. \text{let } z = e \) in \( e_b \).

Proof.

- Let \( z = e \) in \( \lambda x. e_b = \lambda x. \text{let } z = e \) in \( e_b \)
  \[ = \lambda x. \text{let } z = v \text{ in } e_b \]
  \[ = \lambda x. (e_b)[z := v] \]

\[ = \lambda x. \text{let } z = v \text{ in } e_b \]

\[ = \lambda x. \text{let } z = e \text{ in } e_b \]

\[ = \lambda x. \text{let } z = e \text{ in } e_b \]

- Theorem 3. If \( e' \rightarrow^* v \), \( z \) is fresh, and \( x_i \notin \text{FV}(e') \) then let \( z = e' \) in \( \lambda x. e[u := z] = \lambda x. e[u := e'] \).

Proof.

- Let \( z = e' \) in \( \lambda x. e[u := z] = \lambda z = v \) in \( \lambda x. e[u := z] = (\lambda x. e[u := z])[z := v] \)
  \[ = \lambda x. e[u := z][z := v] \]
  \[ = \lambda x. e[u := e'] \]

D. GLOBAL CORRECTNESS THEOREM

Let \( \tilde{X} \) be a collection of sets such that each \( X_i \) is a subset of the set of constants in the interpreter language. Further, \( L(\tilde{X}) \) is also a subset of the set of constants in the interpreter language.

Definition 4. Given a finite collection of sets \( \tilde{X} \), \( L(\tilde{X}) \)

We can then abstractly write an interpreter for \( L(\tilde{X}) \) using a sugared version of the interpreter language as follows.

\[ f : L(\tilde{X}) \times S \times D \rightarrow A \]

\[ f(s, i, t_1) = g_i(h^i_s(i, x_i), h^i_t(d, x_i)) \]

Applying currying, lambda lowering, expression lifting, and quoting to \( f \) yields the sugared term below.

\[ f'(x_i, s) = \lambda z. \text{let } y = h^i_s(i, x_i) \text{ in } [\lambda d. g_i(y, h^i_t(d, x_i))] \]

\[ f'(t_1, s) = \lambda z. \text{let } y = h^i_s(i, x_i), \text{ and } u_1 = f'(t_1, s), \text{ and } \cdots \]

\[ \text{in } [\lambda d. \tilde{g}_i(y, h^i_t(d, x_i), u_1(d), \cdots, u_c(d))] \]

Theorem 4. For any \( t \in L(\tilde{X}) \), for any \( i \in S \), if \( h^i_s \) and \( h^i_t \) are total and quote free, then \( f'(t, s) = [\lambda d. f(t, s, d)] \).

Proof. By structural induction on \( t \).

- Suppose \( t = x_i \). Since \( h^i_s \) is total and quote free, \( h^i_s(i, x_i) = v \) and \( v \notin [c] \).

\[ f'(x_i, s) = \lambda z. \text{let } y = h^i_s(i, x_i) \text{ in } [\lambda d. g_i(y, h^i_t(d, x_i))] \]

\[ = \lambda z. \text{let } y = v \text{ in } [\lambda d. g_i(y, h^i_t(d, x_i))] \]

\[ = [\lambda d. g_i(v, h^i_t(d, x_i))] \]

\[ = [\lambda d. g_i(h^i_s(i, x_i), h^i_t(d, x_i))] \]

\[ = [\lambda d. f(x_i, s, d)] \]
Suppose \( t = c_j(\bar{x}_j, \bar{t}_j) \). Since \( h^v_{i^v_j} \) is total and quote free, \( h^v_{i^v_j}(\bar{s}, \bar{x}_j) = v \) and \( v \neq [c] \).

\[
f'(c_j(\bar{x}_j, \bar{t}_j), \bar{s}) = \text{let } y = h^v_{i^v_j}(\bar{s}, \bar{x}_j), \quad u_1 = f'(t^v_{i^v_j}, \bar{s}),
\]

\[
\vdots
\]

\[
\text{in } [\lambda \bar{d}. h_{i^v_j}(y, h^v_{i^v_j}(\bar{d}, \bar{x}_j), (\bar{u}_1)_{i^v_j}(\bar{d}), \bar{u}_{i^v_j}(\bar{d}))]
\]

\[
= \text{let } y = v, \quad u_1 = [\lambda \bar{d}. f(t^v_{i^v_j}, \bar{s}, \bar{d})],
\]

\[
\vdots
\]

\[
\text{in } [\lambda \bar{d}. h_{i^v_j}(y, h^v_{i^v_j}(\bar{d}, \bar{x}_j), (\bar{u}_1)_{i^v_j}(\bar{d}), \bar{u}_{i^v_j}(\bar{d}))]
\]

\[
= [\lambda \bar{d}. h_{i^v_j}(v, h^v_{i^v_j}(\bar{d}, \bar{x}_j), (\lambda \bar{d}. f(t^v_{i^v_j}, \bar{s}, \bar{d}))(\bar{d}), \bar{u}_{i^v_j}(\bar{d}))], \quad f(t^v_{i^v_j}, \bar{s}, \bar{d})]
\]

\[
= [\lambda \bar{d}. h_{i^v_j}(v, h^v_{i^v_j}(\bar{d}, \bar{x}_j), f(t^v_{i^v_j}, \bar{s}, \bar{d}), \bar{u}_{i^v_j}(\bar{d}))]
\]

\[
= [\lambda \bar{d}. h_{i^v_j}(v, h^v_{i^v_j}(\bar{d}, \bar{x}_j), f(t^v_{i^v_j}, \bar{s}, \bar{d}), \bar{u}_{i^v_j}(\bar{d}))]
\]

\[
= [\lambda \bar{d}. f(c_j(\bar{x}_j, \bar{t}_j), \bar{s}, \bar{d})]
\]

\[
\square
\]

Corollary 1. For any \( t \in L(\bar{X}) \), for any \( \bar{s} \in \bar{S} \), for any \( \bar{d} \in \bar{D} \), if \( h^v_{i^v_j} \) and \( h^v_{i^v_j} \) are total and quote free, then \( \text{eval}(f'(t, \bar{s}))(\bar{d}) = f(t, \bar{s}, \bar{d}) \).