A Transformation-Based Foundation for Semantics-Directed Code Generation

Arthur Nunes-Harwitt

Rochester Institute of Technology, Rochester NY 14623, USA
anh@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 to transform an interpreter into a compiler. An extended example suggests the utility of the technique. Finally, this technique is compared to staging and partial evaluation.

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[9] 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 using those values to perform some of the computation that does not depend on the remaining parameters. Kleene's S-m-n theorem establishes that the minimal form of specialization is computable, so programs can be written that perform this task. Rodney M. Bustall and John Darlington[2] provide the conceptual foundation for partial evaluation in the form of equational reasoning that involves *unfolding*, or expanding definitions, and *folding*, or reducing definitions. These ideas have been successfully used in many instances to manually derive a sophisticated algorithm from a naive one[3]. Much effort has been expended creating *partial evaluators*, or fully automatic programs that perform partial evaluation. To produce code generators, traditional partial evaluators must have the property of being self-applicable. Although partial evaluators have also been successfully used in many instances, they often need hints in the form of *binding 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[5] and related techniques that generate text. While William L. Scherlis developed a form of equational reasoning similar to Bustall and Darlington, subsequently he and Ulrik Jørring [10] 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[13,12,15].

This paper is concerned with an alternative approach to semantics-directed code generation. It makes the following contributions. It identifies a new technique for deriving a code generator from an interpreter in the form of four essential transformations. Motivation is provided for this technique and the transformations are presented formally. An extended example suggests the utility of the technique. Finally, this technique is compared to staging and partial evaluation.

2 Transformation Technique

The motivation for this transformation technique comes from denotational-semantics[14,18]. It is common knowledge that a denotational definition can be understood as an interpreter. Further, the denotational definition can also be understood as a compiler; given a term, we are free to evaluate the recursive calls and derive a λ-term. The rules described below concern answering the question: How can a denotational-style interpreter be modified so that it too generates a λ-term?

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[11], 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.

```
> (m 2)
#<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)))

> (m 2)
(lambda (y) (* 2 y))
```

But consider the following more complicated example of raising b to the nth 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
   `(lambda (b)
       (if (= ,n 0)
           1
           (* b ((p (- ,n 1)) b)))))

> (p 3)
(lambda (b)
   (if (= 3 0) 1 (* b ((p (- 3 1)) 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.

```
; original

; curried

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

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

> (p 3)
(lambda (b) (* b ((p (- 3 1)) 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 λ-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.

```
; original

; curried

; lambda lowered

(define (p n) ; expression lifted
   (if (= n 0)
       (lambda (b) 1)
       (let ((f (p (- n 1))))
          (lambda (b) (* b (f b))))))

(define (p n) ; quoted
   (if (= n 0)
       `(lambda (b) 1)
       (let ((f (p (- n 1))))
          `(lambda (b) (* b (,f b))))))
```

```
> (p 3)
(lambda (b)
   (* b ((lambda (b)
            (* b ((lambda (b)
                     (* b ((lambda (b) 1) b)))
                   b)))
         b)))
```

Although the result is hard to read, a respectable Scheme compiler should generate efficient code from the resulting term. Although ideally the generated code would be more readable, we can make it more pleasant looking by post-processing with copy-propagation and dead-code elimination.

```
(lambda (b) (* b (* b (* b 1))))
```

2.5 Summary and Formalization

These examples illustrate how a procedure can be modified so that it generates a λ-term. There are two key ideas: (*i*) An expression within an abstraction cannot be evaluated, and so the code is restructured so that the expression is no longer within the abstraction. (*ii*) For such restructuring to be plausible, the expression in question cannot contain variables that are the abstraction's formal parameters.

$$\begin{array}{rl}
\text{Terms } e & ::= c \\
 & ::= x \\
 & ::= ,y \\
 & ::= (\lambda \bar{x}.e) \\
 & ::= [\![e]\!] \\
 & ::= (e_0\ \bar{e}) \\
 & ::= \mathsf{let}\ ,y = e\ \mathsf{in}\ e' \\
 & ::= \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
\text{Values } v & ::= c \\
 & ::= (\lambda \bar{x}.e) \\
 & ::= [\![e]\!]
\end{array}$$

$\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is syntactic sugar.

$(\mathsf{eval}\ e)$ is syntactic sugar.

Fig. 1. Interpreter language syntax.

We model Scheme via a call-by-value λ-calculus with quotation (see figure 1). A new kind of variable, the comma variable, is used together with a let-form to model Scheme's unquote. A let not involving a comma variable is understood

$$\lambda\bar{s},\bar{d}.e \hookrightarrow \lambda\bar{s}.\lambda\bar{d}.e \tag{1}$$

$$(e_c\ \bar{e_s}, \bar{e_d}) \hookrightarrow ((e_c\ \bar{e_s})\ \bar{e_d}) \tag{2}$$

if e_c is an expression that reduces to a curried function.

$$\lambda\bar{x}.\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \hookrightarrow \mathsf{if}\ e\ \mathsf{then}\ (\lambda\bar{x}.e_1)\ \mathsf{else}\ (\lambda\bar{x}.e_2) \quad \text{if } x_i \notin \mathsf{FV}(e) \tag{3}$$

$$\lambda\bar{x}.\mathsf{let}\ z = e\ \mathsf{in}\ e_b \hookrightarrow \mathsf{let}\ z = e\ \mathsf{in}\ \lambda\bar{x}.e_b \quad \text{if } x_i \notin \mathsf{FV}(e) \text{ and } z \neq x_i \tag{4}$$

$$\lambda\bar{x}.e'[u\ :=\ e] \hookrightarrow \mathsf{let}\ z = e\ \mathsf{in}\ \lambda\bar{x}.e'[u\ :=\ z] \quad \text{if } z \text{ is fresh and } x_i \notin \mathsf{FV}(e) \tag{5}$$

$$\begin{aligned}
&\mathsf{let}\ z_1 = e_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ z_n = e_n\ \mathsf{in}\ \lambda\bar{d}.e \hookrightarrow \\
&\mathsf{let}\ \ ,z_1\ =\ e_1\ \mathsf{in}\ \cdots \\
&\quad \mathsf{let}\ \ ,z_n\ =\ e_n\ \mathsf{in} \\
&\qquad [\![\lambda\bar{d}.e[z_1\ :=\ ,z_1]\cdots[z_n\ :=\ ,z_n]]\!] \\
&\text{when } \mathsf{FV}(\lambda\bar{d}.e) = \{z_1,\ldots,z_n\}\text{, each } ,z_i \text{ is fresh,} \\
&\text{and } \mathsf{FV}(e_i) \cap \{z_1,\ldots,z_n\} = \varnothing \text{ for each } i
\end{aligned} \tag{6}$$

Fig. 2. Transformations

in the usual way to abbreviate the application of an abstraction. The eval operator can be defined in terms of the let with comma variables. The semantics is similar to the λ-calculus with quotation found in [13] (see appendix A). 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 lowering. 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 e_2; otherwise the body is e_1. 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 e_b. The let on the right-hand-side substitutes v for z in the abstraction, but it passes right through and becomes a substitution in e_b since z is distinct from the formal parameters.

Rule (5) is expression lifting. 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.

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 sub-structure 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. Here we merely assert 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.

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

- *The empty string. (*`null?`*)*
- *A character in the alphabet. (*`char?`*)*
- *The union of two simple regular expressions. (*`or?`*)*
- *The concatenation of two simple regular expressions. (*`cat?`*)*

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`).

```
(define (match regexp cl k)
  (cond ((null? regexp) (k cl))
        ((char? regexp)
         (if (null? cl)
             #f
             (and (eq? (car cl) regexp) (k (cdr cl)))))
        ((or? regexp)
         (or (match (exp1<-or regexp) cl k)
             (match (exp2<-or regexp) cl k)))
        ((cat? regexp)
         (match (exp1<-cat regexp)
                cl
                (lambda (cl2) (match (exp2<-cat regexp) cl2 k))))
        (else (error 'match "match's first input is not a regexp"))))
```

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))))
          (else (error 'match1 "match1's input is not a regexp")))))
```

3.2 Lambda-lowering

Since `(cond (`e_1 e_2`) ...)` $\equiv$ `(if` e_1 e_2 `(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-expression[1].

```
(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)
         (lambda (cl k)
           (or ((match2 (exp1<-or regexp)) cl k)
               ((match2 (exp2<-or regexp)) cl k))))
        ((cat? regexp)
         (lambda (cl k)
           ((match2 (exp1<-cat regexp))
            cl
```

[1] 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.

```
           (lambda (cl2) ((match2 (exp2<-cat regexp)) cl2 k)))))
        (else (error 'match2 "match2's input is not a regexp"))))
```

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 k))))))
        (else (error 'match3 "match3's input is not a regexp"))))
```

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 unquoted in the text.

```
(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 k))))))
        (else (error 'match4 "match4's input is not a regexp"))))
```

3.5 Output

When the regular expression is $\mathtt{a}(\mathtt{a} \cup \mathtt{b})$, the simplified output becomes the following.

```
(lambda (cl k)
  (let ((k (lambda (cl2)
             (or (if (null? cl2)
                     #f
                     (and (eq? (car cl2) #\a) (k (cdr cl2))))
                 (if (null? cl2)
                     #f
                     (and (eq? (car cl2) #\b) (k (cdr cl2))))))))
    (if (null? cl) #f (and (eq? (car cl) #\a) (k (cdr cl))))))
```

4 Comparison to Other 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[6].

The Futamura projections concern the following observations. We model a programming language $\mathcal{L}$ as a three-tuple $\mathcal{L} = \langle \mathcal{E}, L, D \rangle$, where $L \subseteq D$ and $\mathcal{E} : L \times D^* \to D$ maps programs and inputs to output. Given a programming language $\mathcal{L}$, a partial evaluator m has the property that for any $\mathcal{L}$ program e, $(\mathcal{E}\ e\ (d_1, d_2)) = (\mathcal{E}\ (\mathcal{E}\ m\ (e, d_1))\ d_2)$. Now first observe that if e is an interpreter for $\mathcal{L}_2$ and p is an $\mathcal{L}_2$ program, then $(\mathcal{E}\ e\ (p, d)) = (\mathcal{E}\ (\mathcal{E}\ m\ (e, p))\ d)$. Thus $(\mathcal{E}\ m\ (e, p))$ can be regarded as the target code, and $\lambda p.(\mathcal{E}\ m\ (e, p))$ can be regarded as a compiler. Second, observe that $(\mathcal{E}\ m\ (e, p)) = (\mathcal{E}\ (\mathcal{E}\ m\ (m, e))\ p)$. Thus we can reify the previous compiler abstraction as $(\mathcal{E}\ m\ (m, e))$; i.e., the partial evaluator is applied 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[1,16] 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 often 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 them appears straightforward; the biggest difficulty seems to be verifying that particular terms terminate. Common cases involving structure predicates and substructure selectors should be verifiable.

There are interpreters for which the transformation technique does not succeed. That is also the case for partial evaluation algorithms[2]. Coming up with the right binding time improvements to help a partial evaluator can be challenging because partial evaluation algorithms are quite complicated[4]. 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.

In this spirit, we consider a variation on the extended example in section 3. Suppose we would like to add Kleene star to the regular expression interpreter. We start by augmenting the interpreter in the following naive fashion.

```
...
((star? regexp)
 (or (k cl)
     (match (exp<-star regexp)
            cl
            (lambda (cl3)
               (if (eq? cl cl3) #f (match regexp cl3 k))))))
```

PGG[17,16] is a partial evaluator that follows the cogen approach. According to Neil D. Jones[8], it is one of the most sophisticated partial evaluation programs available at the time of this writing. Indeed, it has no trouble partially evaluating this augmented example on a static regular expression that includes a Kleene star form. Nevertheless, it seems to be missing the key feature of making the generator available as stand-alone code; a compiler based on PGG must include the PGG environment.

[2] Early partial evaluators had trouble with assignment and/or higher-order functions. More recent partial evaluators have overcome these and other obstacles.

The transformation technique does not succeed on this augmented interpreter. If we apply currying, lambda lowering, and start to apply expression lifting, it becomes apparent that one expression cannot be lifted because it will not terminate outside the λ-expression.

```
...
((star? regexp)
 (let ((f1 (match25 (exp<-star regexp))))
   (lambda (cl k)
     (or (k cl)
         (f1
          cl
          (lambda (cl3) ; if lifted,  (match25 regexp)  will loop!
            (if (eq? cl cl3) #f ((match25 regexp) cl3 k))))))))
```

It is clear that the expression `(match25 regexp)` will loop if lifted. This problem is similar to a problem encountered by Gunter[7] when turning an operational semantics into a denotational semantics. We adopt his solution here. Let $f_2 =$ `(match25 regexp)`, then `(match25 regexp)` $= \lambda(c\ell, k). \cdots$ `(match25 regexp)` $\cdots$ becomes $f_2 = \lambda(c\ell, k). \cdots f_2 \cdots$, at which point we consider the fixed point solution of f_2. We then get the following code.

```
...
((star? regexp)
 (let ((f1 (match3 (exp<-star regexp))))
   (letrec ((f2 (lambda (cl k)
                  (or (k cl)
                      (f1
                       cl
                       (lambda (cl3)
                         (if (eq? cl cl3) #f (f2 cl3 k))))))))
     f2)))
```

The body of the let is not what the quoting rule needs, and so we eta-expand.

```
...
((star? regexp)
 (let ((f1 (match35 (exp<-star regexp))))
   (lambda (cl k)
     ((letrec ((f2 (lambda (cl k)
                     (or (k cl)
                         (f1
                          cl
                          (lambda (cl3)
                            (if (eq? cl cl3) #f (f2 cl3 k))))))))
        f2) cl k))))
```

Now the quoting rule can be applied. When performed, we get a code generator for regular expressions that includes Kleene star forms.

5 Conclusion

This paper has presented a new transformation technique for deriving a code generator from an interpreter. Further, it has provided an example that illustrates the ideas. Finally, it argues 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 2 can be effectively applied to any denotational-style interpreter, it is not yet clear to what extent that class of interpreters can be extended. Even the longest example considered in this paper is 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 future publication. 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.

6 Acknowledgements

I would like to thank Axel Schreiner for carefully reading previous drafts of this paper, and Matthew Fluet for reading a draft and for looking carefully at my proofs.

References

1. L. Birkedal, M. Welinder. Hand-Writing Program Generator Generators. *Programming Language Implementation and Logic Programming*, Springer, 1994.
2. R. M. Bustall, J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, Vol. 24, No. 1, 44–67, 1977.
3. O. Danvy, H. K. Rohde. On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation. *Information Processing Letters*, Vol. 99, No. 4, 158–162, 2006.
4. O. Danvy and R. Vestergaard. Semantics Based Compiling: A Case Study in Type Directed Partial Evaluation. *Eighth International Symposium on Programming Language Implementation and Logic Programming*, 182–497, 1996.
5. M. Feeley, G. LaPalme. Using Closures for Code Generation. *Computer Language*, Vol. 12, No. 1, 47–66, 1987.
6. Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, Vol. 2, No. 5, 45–50, 1971.
7. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.
8. N. D. Jones. Personal Communication.
9. N. D. Jones, C. K. Gomard, P. Sestoft, L. O. Andersen, T. Mogensen. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.
10. U. Jørring, W. L. Scherlis. Compilers and Staging Transformations. *Symposium on Principles of Programming Languages*, 86–96, 1986.

11. R. Kelsey, W. Clinger, J. Rees editors. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, 26–76, 1998.
12. E. Moggi, W. Taha, Z. Benaissa, T. Sheard. An Idealized MetaML: Simpler, and More Expressive. *Proeedings of the European Symposium on Programming*, 193–207, 1999.
13. A. Nanevski, F. Pfenning. Staged Computation with Names and Necessity. *Journal of Functional Programming*, Vol. 15, No. 6, 893–939, 2005.
14. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1981.
15. W. Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, Hillsboro, Oregon, USA, 1999.
16. P. J. Thiemann. Cogen in Six Lines. *ACM SIGPLAN Notices*, Vol. 31, No. 6. ACM, 1996.
17. P. J. Thiemann. *The PGG system–user manual.* 2000.
18. F. Turbak, D. Gifford, M. A. Sheldon. *Design Concepts in Programming Languages.* MIT Press, 2008.

A Operational Semantics

$$\frac{\delta(c_{op}, \bar{v}) = v'}{(c_{op}\ \bar{v}) \rightarrow v'} \qquad \frac{}{((\lambda \bar{x}.e)\ \bar{v}) \rightarrow e[\bar{x}\ :=\ \bar{v}]}$$

$$\frac{v \neq [\![e]\!]}{\mathsf{let}\ ,y = v\ \mathsf{in}\ e_b \rightarrow e_b[,y\ :=\ v]} \quad \frac{}{\mathsf{let}\ ,y = [\![e]\!]\ \mathsf{in}\ e_b \rightarrow e_b[,y\ :=\ e]}$$

$$\frac{}{\mathsf{if\ False\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_2} \qquad \frac{v \neq \mathsf{False}}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_1}$$

$$\frac{e \rightarrow e'}{(v_1 \cdots v_m\ e\ e_1 \cdots e_n) \rightarrow (v_1 \cdots v_m\ e'\ e_1 \cdots e_n)}$$

$$\frac{e \rightarrow e'}{\mathsf{let}\ ,y = e\ \mathsf{in}\ e_b \rightarrow \mathsf{let}\ ,y = e'\ \mathsf{in}\ e_b} \quad \frac{e \rightarrow e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightarrow \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}$$

B Term Equality

$$\frac{}{e = e}\ \text{reflexive} \quad \frac{e = e'}{e' = e}\ \text{symmetric} \quad \frac{e = e'\ \ e' = e''}{e = e''}\ \text{transitive}$$

$$\frac{e \rightarrow e'}{e = e'}\ \text{reduction} \qquad \frac{e \equiv_\alpha e'}{e = e'}\ \alpha \qquad \frac{e = e'}{\lambda \bar{x}.e = \lambda \bar{x}.e'}\ \xi$$

$$\frac{e = e'}{(e_1 \cdots e_m\ e\ e_{m+1} \cdots e_n) = (e_1 \cdots e_m\ e'\ e_{m+1} \cdots e_n)}$$

$$\frac{e = e'}{\mathsf{let}\ ,y = e\ \mathsf{in}\ e_b = \mathsf{let}\ ,y = e'\ \mathsf{in}\ e_b} \qquad \frac{e = e'}{\mathsf{let}\ ,y = e''\ \mathsf{in}\ e = \mathsf{let}\ ,y = e''\ \mathsf{in}\ e'}$$

$$\frac{e = e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \quad \frac{e = e'}{\mathsf{if}\ e_0\ \mathsf{then}\ e\ \mathsf{else}\ e_2 = \mathsf{if}\ e_0\ \mathsf{then}\ e'\ \mathsf{else}\ e_2}$$

$$\frac{e = e'}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e'} \qquad \frac{e = e'}{[\![e]\!] = [\![e']\!]}$$

C Correctness Theorems

Theorem 1 *If* $e \to^* v$, *and* $x_i \notin \mathsf{FV}(e)$ *then* $\lambda\bar{x}.\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 = \mathsf{if}\ e\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2$.

Proof. By case analysis on v.

– *Suppose* $v \neq \mathsf{False}$.

$$\begin{aligned}
\lambda\bar{x}.\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &= \lambda\bar{x}.\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\\
&= \lambda\bar{x}.e_1\\
&= \mathsf{if}\ v\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2\\
&= \mathsf{if}\ e\ \mathsf{then}\ \lambda\bar{x}.e_1\ \mathsf{else}\ \lambda\bar{x}.e_2
\end{aligned}$$

– *Suppose* $v = \mathsf{False}$.
The argument is similar.

□

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

Proof.

$$\begin{aligned}
\mathsf{let}\ z = e\ \mathsf{in}\ \lambda\bar{x}.e_b &= \mathsf{let}\ z = v\ \mathsf{in}\ \lambda\bar{x}.e_b\\
&= (\lambda\bar{x}.e_b)[z\ :=\ v]\\
&= \lambda\bar{x}.(e_b[z\ :=\ v])\\
&= \lambda\bar{x}.\mathsf{let}\ z = v\ \mathsf{in}\ e_b\\
&= \lambda\bar{x}.\mathsf{let}\ z = e\ \mathsf{in}\ e_b
\end{aligned}$$

□

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

Proof.

$$\begin{aligned}
\mathsf{let}\ z = e'\ \mathsf{in}\ \lambda\bar{x}.e[u\ :=\ z] &= \mathsf{let}\ z = v\ \mathsf{in}\ \lambda\bar{x}.e[u\ :=\ z]\\
&= (\lambda\bar{x}.e[u\ :=\ z])[z\ :=\ v]\\
&= \lambda\bar{x}.(e[u\ :=\ z][z\ :=\ v])\\
&= \lambda\bar{x}.e[u\ :=\ v]\\
&= \lambda\bar{x}.e[u\ :=\ e']
\end{aligned}$$

□