Monads, Indexes and Transformations*

Françoise Bellegarde† and James Hook‡
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999
{bellegar,hook}@cse.ogi.edu

CS/E 92-017

October 12, 1992

Abstract

The specification and derivation of substitution for the de Bruijn representation of λ-terms is used to illustrate programming with a function-sequence monad. The resulting program is improved by automatable program transformation methods into an efficient implementation that uses primitive machine arithmetic. These transformations illustrate new techniques that assist the discovery of the arithmetic structure of the solution.

Introduction

Substitution is one of many problems in computer science that, once understood in one context, is understood in all contexts. Why, then, must a different substitution function be written for every abstract syntax implemented? This paper shows how to specify substitution once and use the monadic structure of the specification to instantiate it on different abstract syntax structures. It also shows how to automatically derive an efficient implementation of substitution from this very abstract specification.

Formal methods that support reasoning about free algebras from first principles based on their inductive structure are theoretically attractive because they have simple and expressive

*Submitted to the Colloquium on Formal Approaches of Software Engineering, 1993.

†Bellegarde is currently at Western Washington University, Bellingham, WA 98225.

‡Both authors are supported in part by a grant from the NSF (CCR-9101721).

u within the scope of v in (2) are decremented in (3) because the λ binding v was removed in the contraction.

One way to characterize substitution with the de Bruijn representation is by a series of functions, each to be applied in a different context. The first context is the body of the contracted λ. Here all references to the index 0 are to be replaced by the argument $\lambda\,.\,0\,1$ while all references to global variables are to be decremented:

$$\begin{aligned} \sigma_0 0 &= \lambda\,.\,0\,1 \\ \sigma_0(n+1) &= n \end{aligned}$$

Note that both right hand sides are terms, not simply integers[1]. In the second context, all occurrences of 0 remain unchanged, references to index 1 are now to the argument and references to indexes greater than 1 are global. All occurrences of free variables in the argument must be incremented. This gives:

$$\begin{aligned} \sigma_1 0 &= 0 \\ \sigma_1 1 &= \lambda\,.\,0\,2 \\ \sigma_1(n+2) &= (n+1) \end{aligned}$$

In this case, these are the only substitutions needed, but in general any number may be required. The key to this development is to calculate this sequence of functions and then use a generic recursion scheme, such as that provided by the *map* function, that has been specialized to select the function from the family appropriate to the context.

The first thing to observe about the sequence is that its general shape is:

$$\begin{aligned} \sigma_{i+1} 0 &= 0 \\ \sigma_{i+1}(n+1) &\approx \sigma_i n \end{aligned}$$

To make it exact it is necessary to increment all global variables in $\sigma_i n$ without incrementing the local variables. This is done by another sequence of functions:

$$\begin{aligned} f_0 n &= n+1 \\ f_1 0 &= 0 \\ f_1(n+1) &= n+2 \\ f_2 0 &= 0 \\ f_2 1 &= 1 \\ f_2(n+2) &= n+3 \end{aligned}$$

Observe that in the example a single application of f_1 to the body of $\sigma_1 1$ accounts for $\lambda\,.\,0\,1$ being adjusted to $\lambda\,.\,0\,2$. In general the f_i are generated by:

$$\begin{aligned} f_{i+1} 0 &= 0 \\ f_{i+1}(n+1) &= (f_i n) + 1 \end{aligned}$$

[1]The coercion of numbers to terms implicit here will become explicit in the programs developed below.

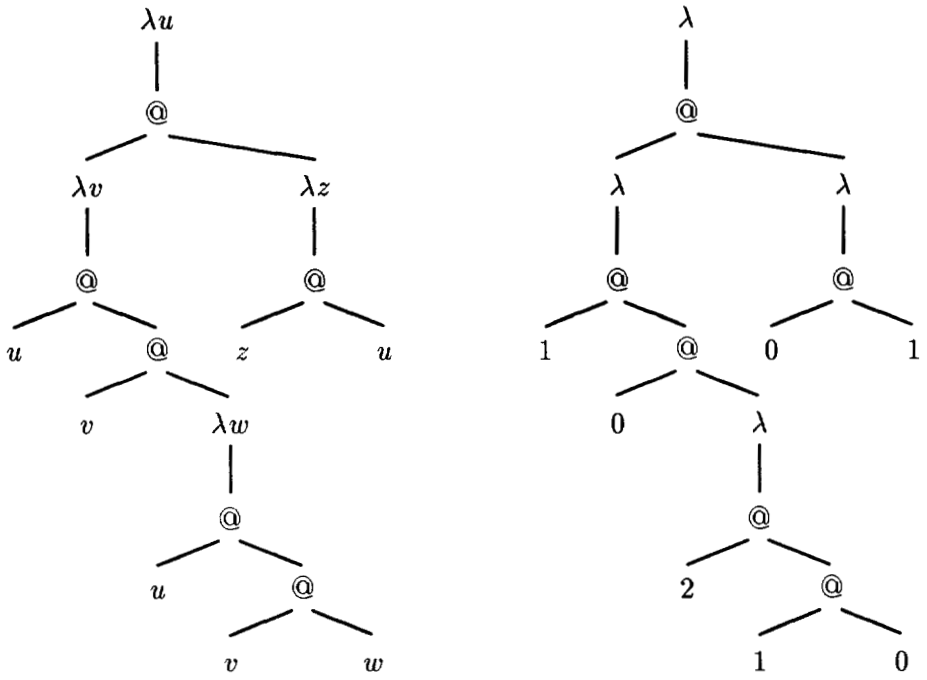

Figure 1: Tree representations of $\lambda u\,.\,(\lambda v\,.\,uv(\lambda w\,.\,uvw))(\lambda z\,.\,zu)$.

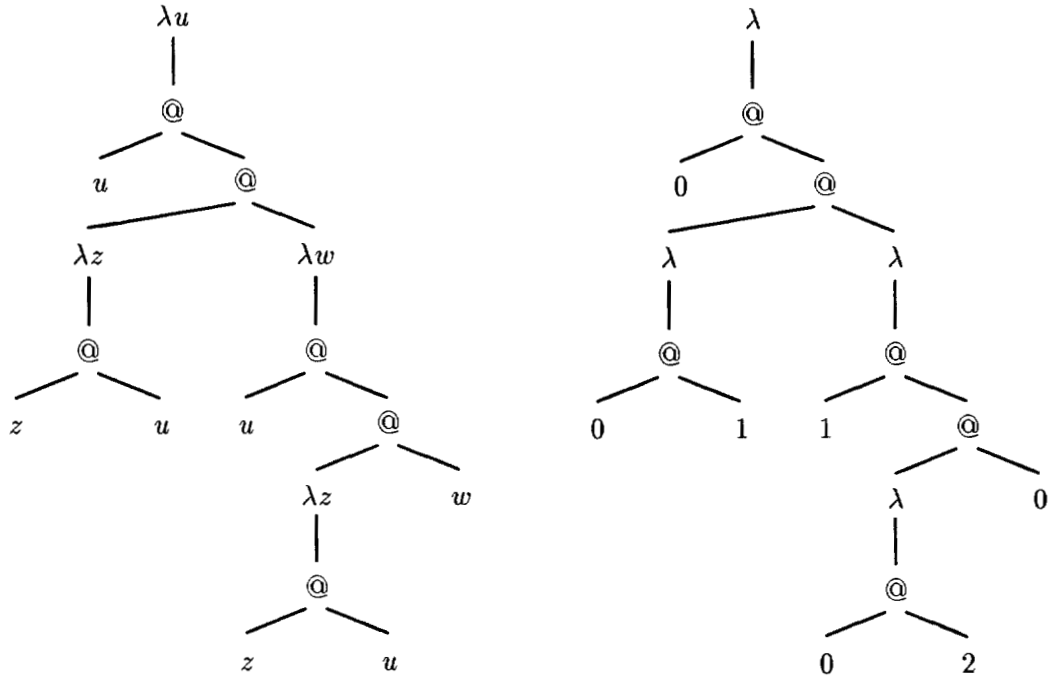

Figure 2: Representations of the contracted term.

So, assuming a *map* that applies a family of functions, the family of substitution functions, $(\sigma_0, \sigma_1, \ldots)$, is given by the initial substitution, σ_0, and the recurrence:

$$\begin{aligned} \sigma_{i+1} 0 &= 0 \\ \sigma_{i+1}(n+1) &= map\,(f_0, f_1, \ldots)\,(\sigma_i n) \end{aligned}$$

Given the sequence of functions, $(\sigma_0, \sigma_1, \ldots)$, mapping indexes to terms, the *map* function for sequences can be used to apply the sequence of substitution functions. This, however, results in terms of terms, since every variable has replaced its index by a term. This is not a problem, however, because the *Term* type constructor developed below is designed to be a monad; monads have a polymorphic function, *mult*, which performs the requisite flattening.

2 Monads

A *monad* is a concept from category theory that has been used to provide structure to semantics[7] and to specifications[10]. In the computer science setting a monad is defined by a parametric data type constructor, T, and three polymorphic functions:

$$\begin{aligned} map &: (\alpha \to \beta) \to T\alpha \to T\beta \\ unit &: \alpha \to T\alpha \\ mult &: TT\alpha \to T\alpha \end{aligned}$$

The *map* function is required to satisfy:

$$\begin{aligned} map\ id_\alpha &= id_{T\alpha} \\ map\,(f \circ g) &= map\ f \circ map\ g \end{aligned}$$

The polymorphic functions *unit* and *mult* must satisfy:

$$\begin{aligned} mult_\alpha \circ unit_{T\alpha} &= id_{T\alpha} \\ mult_\alpha \circ (map\ unit_\alpha) &= id_{T\alpha} \\ mult_\alpha \circ mult_{T\alpha} &= mult_\alpha \circ (map\ mult_\alpha) \end{aligned}$$

A simple example of a monad is *list*. For lists, *map* is the familiar `mapcar` function of Lisp, *unit* is the function that produces a singleton list, and *mult* is the concatenate function that flattens a list of lists into a single list. Other examples of monads are given by Wadler[10].

Several categorical concepts are implicit above. The functional programming category has types as objects and (computable) functions as arrows. (Values are viewed as constant functions—arrows from the one element type.) The requirements on *map* specify that the type constructor T and the *map* function together define a *functor*. The polymorphic types of *unit* and *mult* implicitly require them to be *natural transformations*. The three laws given for them are the *monad laws*.

Monads have been used to structure specifications (and semantics) because it is often possible to characterize interesting facets of a specification as a monad. Algorithms to exploit the particular facet may frequently be expressed in terms of the *map*, *unit* and *mult* functions with no explicit details of the type constructors. Finally, the many facets are brought together by composing the type constructors.

3 The term monad

The development in Section 1 suggests that the specification of the substitution operation will be straightforward in a monadic data type with an appropriate *map*. To be monadic, the data type must be parametric. The following simple type declaration is sufficient[2]:

$$\begin{array}{lcl} \textbf{datatype}\ \mathit{Term}(\alpha) & = & \mathit{Var}(\alpha) \\ & | & \mathit{Abs}(\mathit{Term}(\alpha)) \\ & | & \mathit{App}(\mathit{Term}(\alpha) * \mathit{Term}(\alpha)) \end{array}$$

Using techniques developed in earlier work, it is possible to automatically generate *map*, *mult* and *unit* functions for this type realizing a monadic structure[5]. Unfortunately, the *map* function obtained with those techniques does not work with families of functions.

To accommodate the function sequences a new category, FUNSEQ, is used. The objects are data types, as before, but the morphisms are sequences of functions (formally $\mathbf{HOM}(A,B) = (B^A)^\omega$). Identities are constant sequences of identities from the underlying category; composition is pointwise, i.e. $(f_i)_{i\in\omega} \circ (g_i)_{i\in\omega} = (f_i \circ g_i)_{i\in\omega}$.

The *map* function for *Term* exploits the new structure by shifting the series of functions whenever it enters a new context. Its definition is given as a functional program:

$$\begin{array}{lcl} \mathit{map}\,(f_0, f_1, \ldots)\,(\mathit{Var}\,x) & = & \mathit{Var}((f_0, f_1, \ldots)x) \\ \mathit{map}\,(f_0, f_1, \ldots)\,(\mathit{Abs}\,t) & = & \mathit{Abs}(\mathit{map}\,(f_1, f_2, \ldots)\,t) \\ \mathit{map}\,(f_0, f_1, \ldots)\,(\mathit{App}(t,t')) & = & \mathit{App}(\mathit{map}\,(f_0, f_1, \ldots)\,t, \mathit{map}\,(f_0, f_1, \ldots)\,t') \end{array}$$

It is easily verified that $(\mathit{Term}, \mathit{map})$ satisfy the categorical definition of a functor.

Looking at these definitions, it is clear how to insert an ordinary function or value into the category, and it is straightforward to insert the families of functions needed for the example by giving the initial element of the sequence and the functional that generates all others. However, it is also necessary to define the mapping that pulls a computation from FUNSEQ back into the category of functional programs. This is accomplished by simply taking the first element of the function sequence. Thus, one way to realize the *map* function of FUNSEQ in a functional programming setting is with the *map_with_policy* function introduced in Hook, Kieburtz and Sheard[5]:

$$\begin{array}{lcl} \mathit{map\_with\_policy}\ Z\ f\ (\mathit{Var}\ x) & = & \mathit{Var}(f x) \\ \mathit{map\_with\_policy}\ Z\ f\ (\mathit{Abs}\ t) & = & \mathit{Abs}(\mathit{map\_with\_policy}\ Z\ (Z f)\ t) \\ \mathit{map\_with\_policy}\ Z\ f\ (\mathit{App}(t,t')) & = & \mathit{App}(\mathit{map\_with\_policy}\ Z\ f\ t, \\ & & \quad \mathit{map\_with\_policy}\ Z\ f\ t') \end{array}$$

[2]This is a simplified form of the `Term` data type in Hook, Kieburtz and Sheard[5].

```
fun apply_substitution σ0 M
  = let fun succ x = x + 1
        fun transform_index f
            = λn . if n = 0 then n else 1 + f(n − 1)
        fun transform_substitution σ
            = λn . if n = 0 then unit 0
                   else map_with_policy transform_index succ (σ(n − 1))
    in  mult(map_with_policy transform_substitution σ0 M)
    end
```

Figure 3: Substitution function

In this encoding Z is the functional that generates the sequence and f is the seed value. That is,

$$(map\,(f, Zf, Z^2f, \ldots))_0 = map\_with\_policy\ Z\ f$$

Note the projection of the first element from the family of functions on the left hand side indicated by the subscript 0.

The name *map_with_policy* refers to the notion of *policy function* introduced by Kieburtz[6, 5]. It refers to a type-specific function, such as Z above, that is embedded into the program for a general polymorphic operator to produce a specialized, monomorphic operator using a similar control scheme.

The *unit* and *mult* functions automatically generated for *Term* can be lifted to FUNSEQ. Their definitions are:

$$\begin{array}{lcl} unit & = & Var \\ mult\ (Var\ x) & = & x \\ mult\ (Abs\ t) & = & Abs(mult\ t) \\ mult\ (App(t, t')) & = & App(mult\ t, mult\ t') \end{array}$$

Simple inductions show that they satisfy the monad laws.

With these definitions in place the complete definition of substitution is given in Figure 3. Note that the algorithm makes no explicit mention of the data constructors. It only uses the information about the type implicit in the definition of *map_with_policy*, *unit* and *mult*.

4 Transformation to a first-order set of equations

To obtain a practical algorithm, the substitution function *apply_substitution* in Figure 3 must be made more efficient. This section shows how this transformation can be done automatically.

Program transformation systems operate on systems of first-order equations. To apply them to the specification of substitution the higher-order facets must be translated into first-order structures. A partial evaluation system is used to accomplish this.

The software allowing a complete automatic transformation is not yet written. The transformations below have been performed with the Schism partial evaluator [4] and the Astre program transformation system [1], which are not yet integrated and do not use the same language.

4.1 Transformation of the *map_with_policy* operator

The first step is to rewrite the program using the *map_with_policy* operator for the type $Term(\alpha)$ as a system of first-order functions. A partial evaluator such as Schism [4] can be used to specialize higher-order functions decreasing their order level. For example, consider the particular function σ_0 in the example in Section 1, and the call *apply_substitution* σ_0. A partial evaluator produces a program that does not contain *apply_substitution* in its full generality; it specializes the definition of *apply_substitution* for the particular constant σ_0. This specialization, called *apply_substitution_*σ_0, does not have a function as an argument, so it is first-order.

Unfortunately, this technique is insufficient for processing calls of *map_with_policy*, which is called twice in the program in Figure 3. The specialization of *map_with_policy* for a particular policy function K and seed function g_0 gives the following function *Mwp_g*:

$$\begin{array}{lcl} Mwp\_g\ (g, Var(n)) & = & Var(g(n)) \\ Mwp\_g\ (g, Abs(t)) & = & Abs(Mwp\_g(K\ g, t)) \\ Mwp\_g\ (g, App(t, t')) & = & App(Mwp\_g(g, t), Mwp\_g(g, t')) \end{array}$$

The function *Mwp_g* has a function as an argument. But if it is specialized for a particular function g_0, the partial evaluator has to specialize the internal call $Mwp\_g(K\ g, t)$; it loops on this attempt. Fortunately, the partial evaluator is able to detect this circumstance, allowing it to select another technique. The alternative technique translates the higher-order functions into a system of first-order functions. This standard encoding, which is due to Reynolds [8], is outlined below.

1. The first step constructs a data type that encodes how the higher-order arguments are manipulated and applied. In this case the functions to be encoded are g_0 and $K\ g$. For the constant function, g_0, a constant C is introduced as a summand in the data type *Func*. The argument $K\ g$ cannot be encoded by a simple constant value because it contains g as a free variable. Since g is a higher-order parameter, it will already be represented by a value of type *Func*. Hence the new constructor, F, representing the application of K, must have type $Func \rightarrow Func$. This gives the data type *Func*

$$\begin{array}{lcl} \textbf{datatype}\ Func & = & C \\ & | & F(Func). \end{array}$$

   The introduction of the data type *Func* is a rediscovery of the sequence of functions $g_0, g_1, \ldots$ because it encodes each function in the family. The function g_0 is encoded by C, and the function g_3, for example, is encoded by $F(F(F(C)))$, which is written F^3.

2. The functions appearing as actual arguments are replaced by their encodings. The argument functions do not exist anymore—they are replaced by first-order data. In the call $Mwp\_g(g_0, M)$, g_0 is no longer a function but a first-order value, $\lceil g_0 \rceil$, of type *Func*. The definition of Mwp_g leads to the new function Mwp_g':

$$\begin{array}{lcl} Mwp\_g'(\lceil g \rceil, Var(n)) & = & Var(\lceil g \rceil(n)) \\ Mwp\_g'(\lceil g \rceil, Abs(t)) & = & Abs(Mwp\_g'(F(\lceil g \rceil), t)) \\ Mwp\_g'(\lceil g \rceil, App(t, t')) & = & App(Mwp\_g'(\lceil g \rceil, t), Mwp\_g'(\lceil g \rceil, t')) \end{array}$$

   But since $\lceil g \rceil$ is not a function, the application $\lceil g \rceil(n)$ is nonsense.

3. To make sense of the applications of functional parameters in the orignal programs "application" functions are introduced. Specifically the function $apply\_g$, defined below, decodes applications of the form $\lceil g \rceil(n)$.

$$\begin{array}{rcl} apply\_g(C, n) & = & g_0(n) \\ apply\_g(F(\lceil g \rceil), n) & = & (K\ \lambda n\ .\ apply\_g(\lceil g \rceil, n))(n)\ . \end{array} \qquad (4)$$

   Note that $apply\_g$ is a first-order function because its argument, $\lceil g \rceil$, is an element of the type *Func*. The partial evaluator unfolds the definition of the policy function K to get a first-order expression of $apply\_g(F(\lceil g \rceil), n)$. The definition of Mwp_g' can be completed into:

$$\begin{array}{lcl} Mwp\_g'(\lceil g \rceil, Var(n)) & = & Var(apply\_g(\lceil g \rceil, n)) \\ Mwp\_g'(\lceil g \rceil, Abs(t)) & = & Abs(Mwp\_g'(F(\lceil g \rceil), t)) \\ Mwp\_g'(\lceil g \rceil, App(t, t')) & = & App(Mwp\_g'(\lceil g \rceil\ t), Mwp\_g'(\lceil g \rceil, t')) \end{array}$$

Recall that this encoding is done with respect to a specific call of *map_with_policy* $Z\ g_0\ M$. In the program in Figure 3 there are two such calls, *map_with_policy transform_index succ*$(\sigma(n-1))$ and *map_with_policy transform_substitution* $\sigma_0\ M$. If the partial evaluator succeeds in the transformation of equation (4), then the new functions corresponding to Mwp_g and $apply\_g$ will constitute a first-order program equivalent to the functions generated by *map_with_policy*. This step of the transformation can be automated using a partial evaluator like Schism.

4.2 Application to *apply_substitution*

Using the preceding techniques, the function *apply_substitution* is successfully transformed into the first-order program in Figure 4. The data type *Subst* and the data type *Fseq* are introduced using the techniques above for the encodings of *transform_index* and *transform_substitution*.

$$\begin{array}{rcl} \textbf{datatype}\ Subst & = & S0 \\ & | & SUBST(Subst) \\ \textbf{datatype}\ Fseq & = & SUCC \\ & | & FSEQ(Fseq) \end{array}$$

```
fun apply_substitution_σ0(M)
    = let  fun
           apply_f(SUCC, n)          = s(n)
           | apply_f(FSEQ(f), n)     = if n = 0 then 0
                                       else s(apply_f(f, n − 1))
           fun
           Mwp_f(f, Var(n))          = Var(apply_f(f, n))
           | Mwp_f(f, Abs(t))        = Abs(Mwp_f(FSEQ(f), t))
           | Mwp_f(f, App(t, t'))    = App(Mwp_f(f, t), Mwp_f(f, t'))
           fun
           apply_σ(S0, n)            = σ0(n)
           | apply_σ(SUBST(σ), n)    = if n = 0 then unit(0)
                                       else Mwp_f(Succ, (apply_σ(σ, n − 1)))
           fun
           Mwp_σ(σ, Var(n))          = Var(apply_σ(σ, n))
           | Mwp_σ(σ, Abs(t))        = Abs(Mwp_σ(SUBST(σ), t))
           | Mwp_σ(σ, App(t, t'))    = App(Mwp_σ(σ, t), Mwp_σ(σ, t'))
      in   mult(Mwp_(σ)(S0,M))
      end
```

Figure 4: First-order Program

These two data types are isomorphic to the data type *Nat*[3] which can be implemented efficiently in the hardware. However, the specialized function Mwp_σ does not exploit the efficient implementation since it uses the (essentially unary) representation of the data type instead. Thus, the function $apply\_\sigma$ must peel off all of the data constructors each time Mwp_σ is applied to $Var(n)$. For example, after three levels of abstraction, σ_3 is represented by $SUBST(SUBST(SUBST(S0)))$. (The same is also true of the function Mwp_f.) To eliminate this inefficiency, which was present in the calling behavior of the original specification, the data types *Subst* and *Fseq* must be changed to the uniform data type *Nat*. This transformation can be performed automatically by Astre. Ultimately the explicit use of *Nat* will facilitate the use of primitive arithmetic in the program.

5 Simple transformations

The following two simple transformations are performed automatically by Astre after introducing new function symbols. The first one introduces indexes to count the level of abstractions. The second replaces the composition of *Mwp* with the function *mult* by a single function. The order of these transformations does not matter; they can be done simultaneously.

For technical reasons recursive definitions of the form:

$$g(n) = \textbf{if } n = 0 \textbf{ then } e_1 \textbf{ else } e_2$$

are manipulated more effectively by Astre in the equivalent form:

$$\begin{aligned} g(0) &= e_1[0/n] \\ g(s(n)) &= e_2[s(n)/n] \end{aligned}$$

The notation $e[e'/x]$ denotes the substitution of expression e' for x in e. This restriction of the form of equations ensures the termination of the rewriting used by Astre to unfold the definition of g.

5.1 Introduction of indexes

The isomorphism between the automatically generated type *Subst* and the natural numbers is made explicit by introducing the function $iso\_\sigma : Nat \rightarrow Subst$:

$$\begin{aligned} \textbf{fun} \quad iso\_\sigma(s(i)) &= SUBST(iso\_\sigma(i)) \\ \mid iso\_\sigma(0) &= S0 \end{aligned}$$

The functions $apply\_\sigma$ and Mwp_σ are replaced by the new functions $\sigma(i,n)$ (for $\sigma_i(n)$) and Mwp_σ', respectively. These functions satisfy:

$$\begin{aligned} \textbf{fun } \sigma(i,n) &= apply\_\sigma(iso\_\sigma(i), n) \\ Mwp\_\sigma' &: Nat * Nat \rightarrow Term(Nat) \\ Mwp\_\sigma'(i,n) &= Mwp\_\sigma(iso\_\sigma(i), n) \end{aligned}$$

[3]The constructors for the data type *Nat* are 0 and s, i.e. **datatype** $Nat = 0 \mid s(Nat)$.

```
fun apply_substitution_σ0(M)
    = let fun
          f(0,n)                  = s(n)
          | f(s(i),0)             = 0
          | f(s(i),s(n))          = s(f(i,n))
          fun
          Mwp_f'(i, Var(n))       = Var(f(i,n))
          | Mwp_f'(i, Abs(t))     = Abs(Mwp_f'(s(i),t))
          | Mwp_f'(i, App(t,t'))  = App(Mwp_f'(i,t), Mwp_f'(i,t'))
          fun
          σ(0,n)                  = σ0(n)
          | σ(s(i),n)             = unit(0)
          | σ(s(i),s(n))          = Mwp_f'(0,σ(i,n))
          fun
          Mwp_σ'(i, Var(n))       = Var(σ(i,n))
          | : Mwp_σ'(i, Abs(t))   = Abs(Mwp_σ'(s(i),t))
          | Mwp_σ'(i, App(t,t'))  = App(Mwp_σ'(i,t), Mwp_σ'(i,t'))
      in  mult(Mwp_σ'(0,M))
      end
```

Figure 5: Program with indexes

Using these new equations, the Astre system implements the data type *Subst* using the data type *Nat*. New functions to implement the data type *Fseq* using *Nat* are also provided to the Astre system which then gives the program in Figure 5. The program in Figure 5 does not improve the performance of the program in Figure 4. However, its explicit use of numbers is key to the improvements presented in the next section.

5.2 Composition step

The transformation continues with a simple (automatic) step that replaces the composition of *mult* with Mwp_σ' by a single function. [4] This is accomplished by introducing a function symbol, *Ewp*, which is equated to the composition of *mult* with Mwp_σ':

$$mult(Mwp\_\sigma'(0,M)) = Ewp(0,M)$$

[4] This composition is often called the *Kleisli star* or *natural extension*. *Ewp* is a mnemonic for extension with policy.

```
fun apply_substitution_σ0(M)
    = let fun
          f(0,n)                = s(n)
          | f(s(i),0)           = 0
          | f(s(i),s(n))        = s(f(i,n))
          fun
          Mwp(i, Var(n))        = Var(f(i,n))
          | Mwp(i, Abs(t))      = Abs(Mwp(s(i),t))
          | Mwp(i, App(t,t'))   = App(Mwp(i,t), Mwp(i,t'))
          fun
          σ(0,n)                = σ0(n)
          | σ(s(i),n)           = unit(0)
          | σ(s(i),s(n))        = Mwp(0,σ(i,n))
          fun
          Ewp(i, Var(n))        = σ(i,n)
          | Ewp(i, Abs(t))      = Abs(Ewp(s(i),t))
          | Ewp(i, App(t,t'))   = App(Ewp(i,t), Ewp(i,t'))
      in  Ewp(0,M))
      end
```

Figure 6: Composed Program

Astre gives the program in Figure 6 which uses neither *mult*, nor *Mwp_σ'* (in the figure, *Mwp_f'* has been renamed *Mwp* to simplify the nomenclature).

6 Transformation of the sequence of the σ functions

The transformations in this section exploit the arithmetic arguments introduced above to replace then expensive and redundant recursive calculations in σ and *Ewp* with index arithmetic.

The function $\sigma(i,n)$ of the program in Figure 6 is a rediscovery of the series of functions $\sigma_i(n)$ of Section 1. To further refine this program a specific instance of *apply_substitution* σ_0 must be specified. In what follows, the substitution function σ_0, needed for the contraction described in Section 1, is used to illustrate the specialization. Recall that σ_0 replaces variables of index 0 with the term $\lambda\,.\,0\,1$, which is represented by $Abs(App(\,Var(0),\,Var(1)))$.

$$\begin{aligned}\sigma_0(0) &= Abs(App(\,Var(0),\,Var(1)))\\ \sigma_0(s(n)) &= unit(n)\end{aligned}$$

Unfolding the above equations yields a complete definition of $\sigma(i,n)$:

$$\begin{aligned}
&\sigma(0,0) = Abs(App(\,Var(0),\,Var(1)))\\
&\sigma(0,s(n)) = unit(n)\\
&\sigma(s(i),0) = unit(0)\\
&\sigma(s(i),s(n)) = Mwp(0,\sigma(i,n)) \qquad (5)
\end{aligned}$$

Since the equational program is complete with respect to $Nat * Nat$, the computation of any instance of $\sigma(i,n)$ results in a ground constructor term. For example, $\sigma(4,2)$ yields:

$$\begin{aligned}
\sigma(s(s(s(s(0)))),s(s(0))) &\rightarrow \qquad (6)\\
Mwp(0,\sigma(s(s(s(0))),s(0))) &\rightarrow \qquad (7)\\
Mwp(0,Mwp(0,\sigma(s(s(0)),0))) &\rightarrow\\
Mwp(0,Mwp(0,Var(0))) &\rightarrow\\
Mwp(0,Var(f(0,0))) &\rightarrow\\
Var(f(0,f(0,0))) &\rightarrow\\
Var(f(0,s(0))) &\rightarrow\\
Var(s(s(0)))
\end{aligned}$$

Rewrites (6) and (7) are unfoldings by equation (5). Computation of any instance of $\sigma(i,n)$ by naturals can begin with unfoldings using equation (5) until a subterm, $\sigma(u,v)$, in which u *and/or* v are equal to 0 is obtained.

This suggests a target program of the form:

$$\begin{aligned}
\sigma(i,n) = \;&\textbf{if } i > n \textbf{ then } e_1\\
&\textbf{else if } i = n \textbf{ then } e_2\\
&\textbf{else } e_3
\end{aligned}$$

where e_1, e_2, and e_3 are expressions. The transformation will be beneficial if these expressions are efficient. This step introduces a form of function definition by a conditional (instead of structural induction) that violates the technical restriction on programs used to assure termination of rewriting as required by the Astre system. Presently, Astre does not perform this part of the transformation. Moreover, the transformation does not directly generate the conditional; instead it generates the complete definition:

$$\begin{aligned}
&\sigma(s(i)+k,k) = u_1\\
&\sigma(k,k) = u_2\\
&\sigma(k,s(n)+k) = u_3
\end{aligned}$$

6.1 First transformation step

The general strategy of the two transformation steps that follow is to discover arithmetic operations implicit in the recursion structure of programs. The first step in this process is a definition that makes the iteration structure of functions explicit.

Definition 1 *Let x be a variable of type α, let y_i be a term of type β_i for each $i = 1, \cdots, n$, and let φ be a function of type $\beta_1 * \cdots * \alpha * \cdots * \beta_n \rightarrow \alpha$. The function $\hat{\varphi}$ of type $Nat * (\beta_1 * \cdots * \alpha * \cdots * \beta_n) \rightarrow \alpha$ is defined by:*

$$\begin{aligned} \hat{\varphi}(s(k), (y_1, \cdots, x, \cdots, y_n)) &= \varphi(y_1, \cdots, \hat{\varphi}(k, (y_1, \cdots, x, \cdots, y_n)), \cdots, y_n) \\ \hat{\varphi}(0, (y_1, \cdots, x, \cdots, y_n)) &= x \end{aligned}$$

Proposition 1

$$\hat{\varphi}(k, (y_1, \cdots, \varphi(y_1, \cdots, y, \cdots, y_n), \cdots, y_n) = \varphi(y_1, \cdots, \hat{\varphi}(k, y_1, \cdots, y, \cdots, y_n)), \cdots, y_n))$$

Proof: By induction on k. □

An immediate consequence of Definition 1 is

$$\hat{\varphi}(1, x) = \varphi(x)$$

where $x : \beta_1 * \cdots * \alpha * \cdots * \beta_n$.

Having made the iteration structure of functions explicit, the next theorem helps program transformations exploit that structure. To simplify the exposition, consider the case in which $\varphi : \alpha \rightarrow \alpha$. In this case $\hat{\varphi} : Nat * \alpha \rightarrow \alpha$ and $\hat{\varphi}(k, n) = \varphi^k(x)$, where φ^k denotes k applications of φ. Suppose now that $f : Nat * Nat \rightarrow \alpha$ satisfies the equation: $f(s(i), s(n)) = \varphi(f(i, n))$; then $f(4, 7) = \varphi^4(f(0, 3)) = \hat{\varphi}(4, f(0, 3))$. More generally, $f(i + k, n + k) = \hat{\varphi}(k, f(i, n))$, which is the result expressed by Theorem 1.

Theorem 1 *Assume f of type $Nat^n \rightarrow \alpha$, let y_i be a term of type β_i for each $i = 1, \cdots, n$, and let φ be a function of type $\beta_1 * \cdots * \alpha * \cdots * \beta_n \rightarrow \alpha$. The following are equivalent:*

1. $f(s(x_1), \cdots, s(x_n)) = \varphi(y_1, \cdots, f(x_1, \cdots, x_n), \cdots, y_m)$
2. $\hat{\varphi}(k, (y_1, \cdots, f(x_1, \cdots, x_n), \cdots, y_n)) = f(x_1 + k, \cdots, x_n + k)$

Proof: That *1* implies *2* is obvious by instantiating k to 1. The converse is proved by induction on k. □

To apply this theorem to equation (5), let $Mwp0(x)$ be $Mwp(0, x)$ and introduce the equation:

$$\widehat{Mwp0}(k, \sigma(i, n))) = \sigma(i + k, n + k)$$

This gives the equational definition of $\sigma(i, n)$:

$$\begin{aligned} &\sigma(s(i) + k, k) = \widehat{Mwp0}(k, unit(0)) \\ &\sigma(k, k) = \widehat{Mwp0}(k, Abs(App(Var(0), Var(1)))) \\ &\sigma(k, s(n) + k) = \widehat{Mwp0}(k, unit(n)) \end{aligned}$$

This definition is equivalent to the program below, which is of the form described at the beginning of this section:

$$\begin{aligned}\sigma(i,n) = \;& \textbf{if } i > n \textbf{ then } e_1 \\ & \textbf{else if } i = n \textbf{ then } e_2 \\ & \textbf{else } e_3\end{aligned}$$

where

$$\begin{aligned}e_1 &= \widehat{Mwp0}(n, unit(0)) \\ e_2 &= \widehat{Mwp0}(i, Abs(App(\,Var(0),\, Var(1)))) \\ e_3 &= \widehat{Mwp0}(i, unit(n-i-1))\end{aligned}$$

6.2 Second transformation step

The second transformation step transforms the expressions e_1, e_2 and e_3. The definition of $\widehat{Mwp0}$ of type $Term \rightarrow Term$, obtained by Definition 1, refers to the (inefficient) function *Mwp0*. To get an efficient program an alternative (but equivalent) definition of $\widehat{Mwp0}$ that does not refer to *Mwp0* must be generated. Theorem 2 addresses this issue.

To introduce Theorem 2, consider the function *upto*. Informally, $upto(i,n) = [i, i+1, \cdots, n]$. The function *upto* satisfies the equation $upto(s(i), s(n)) = map\ s\ upto(i,n)$. Let *map_s* be the specialization of the definition of *map* by s:

$$\begin{aligned}&map\_s\ [\,] = [\,] \\ &map\_s\ x :: xs = s(x) :: (map\_s\ xs)\end{aligned}$$

The operators [] and :: are the constructors of the data type $List(\alpha)$. By Theorem 1,

$$(\widehat{map\_s})\,(k, upto(i,n)) = (map\_s)^k\,(upto(i,n)) = upto(i+k, n+k)$$

Theorem 2 will yield the following recursive definition of $(map\_s)^k$, (that is of $\widehat{map\_s}$); it does not refer to *map_s*.

$$\begin{aligned}&(map\_s)^k\ [\,] = [\,] \\ &(map\_s)^k\ x :: xs = s^k(x) :: ((map\_s)^k\ xs)\end{aligned}$$

Note, in this definition $(map\_s)^k$ is the function being defined. It is to be regarded atomically; *map_s* is neither defined nor referred to.

Theorem 2 *Let y_i be a term of type β_i for each $i = 1, \cdots, n$, let φ be a function of type $\beta_1 * \cdots * \alpha * \cdots * \beta_n \rightarrow \alpha$, and let C be a constructor of type α. The following are equivalent:*

1. $\varphi(y_1, \cdots, C(x_1, \cdots, x_n), \cdots, y_n) = C(\varphi_1(x_1), \cdots, \varphi_n(x_n))$
2. $\hat{\varphi}(k, (y_1, \cdots, C(x_1, \cdots, x_n), \cdots, y_n)) = C(\widehat{\varphi_1}(k, x_1), \cdots, \widehat{\varphi_n}(k, x_n))$

Proof: That *1* implies *2* is obvious by instanciating k to 1. The converse is proved by induction on k. □

If C is a constructor of arity zero, Theorem 2 degenerates to the two equations $\varphi(y_1, \cdots, C, \cdots, y_n) = C$ and $\hat{\varphi}(k, (y_1, \cdots, C, \cdots, y_n)) = C$.

To apply this result to $\widehat{Mwp0}$, recall that $Mwp0(x) = Mwp(0, x)$ and that:

$$\begin{array}{lcl} Mwp(i,\ Var(n)) & = & Var(f(i,n)) \\ Mwp(i, Abs(t)) & = & Abs(Mwp(s(i), t)) \\ Mwp(i, App(t,t')) & = & App(Mwp(i,t), Mwp(i,t')). \end{array}$$

Introduction of the specializations $f_0(x) = f(0, x)$, and $Mwp1(x) = Mwp(1, x)$ allows the application of Theorem 2, producing:

$$\begin{array}{lcl} \widehat{Mwp0}(k,\ Var(n)) & = & Var(\widehat{f_0}(k,n)) \\ \widehat{Mwp0}(k, Abs(t)) & = & Abs(\widehat{Mwp1}(k,t)) \\ \widehat{Mwp0}(k, App(s,t)) & = & App(\widehat{Mwp0}(k,s), \widehat{Mwp0}(k,t)). \end{array}$$

It is easy to show that $\widehat{f_0} = \hat{s}$ because $f(0, x) = s(x)$, and that $\hat{s}(k, a) = a + k$ by induction on k. Therefore

$$\begin{array}{rcl} \widehat{Mwp0}(k,\ Var(n)) & = & Var(\widehat{f_0}(k,n)) \\ & = & Var(\hat{s}(k,n)) \\ & = & Var(n+k). \end{array}$$

Although this appears to have progressed, it is incomplete because $\widehat{Mwp1}$ is still defined in terms of *Mwp1*. Attempts to define $\widehat{Mwp1}$ by this method, however, will require the function $\widehat{Mwp2}$; this would continue forever. Fortunately, there is another way in which Theorem 1 may be applied to equation (5), yielding the equation:

$$\widehat{Mwp}(k, (0, \sigma(i,n))) = \sigma(i+k, n+k)$$

Applying the same transformation as above produces:

$$\begin{array}{rl} \sigma(i,n) = & \textbf{if } i > n \textbf{ then } e_1 \\ & \textbf{else if } i = n \textbf{ then } e_2 \\ & \textbf{else } e_3 \end{array}$$

where

$$\begin{array}{l} e_1 = unit(n) \\ e_2 = \widehat{Mwp}(i, (0, Abs(App(\ Var(0),\ Var(1))))) \\ e_3 = unit(n-1). \end{array}$$

Application of Theorem 2 produces a recursive definition of $\widehat{Mwp}$ that does not refer to Mwp:

$$\begin{aligned}
&\widehat{Mwp}(k,(i,\mathit{Var}(n))) = \mathit{Var}(\hat{f}(k,(i,n))) \\
&\widehat{Mwp}(k,(i,\mathit{App}(s,t))) = \mathit{App}(\widehat{Mwp}(k,(i,s)),\widehat{Mwp}(k,(i,t))) \\
&\widehat{Mwp}(k,(i,\mathit{Abs}(t))) = \mathit{Abs}(\widehat{Mwp}(k,(s(i),t)))
\end{aligned} \tag{8}$$

The transformation is not yet finished. Equation (8) remains to be improved by finding a recursive definition of $\hat{f}$ that does not refer to the function f.

6.3 Transformation of $\hat{f}$

Recall the equations for f:

$$f(0,n) = s(n) \tag{9}$$
$$f(s(i),0) = 0 \tag{10}$$
$$f(s(i),s(n)) = s(f(i,n)) \tag{11}$$

Applying Theorem 2 to equation (11) yields:

$$\hat{f}(k,(s(i),s(n))) = s(\hat{f}(k,(i,n)))\,. \tag{12}$$

This suggests attempting a conditional definition for $\hat{f}$. Using equations (9), (10), (11), Theorem 2, Theorem 1, and Definition 1 produces:

$$\hat{f}(k,(0,s(n))) = s(\hat{s}(k,n)) = s(n+k) \tag{13}$$
$$\hat{f}(k,(s(i),0)) = 0 \tag{14}$$
$$\hat{f}(k,(0,0)) = k \tag{15}$$

Applying Theorem 1 to Equation (12) gives:

$$\begin{aligned}
\hat{f}(k,(i+p,n+p)) &= \hat{s}(p,\hat{f}(k,(i,n))) \\
&= \hat{f}(k,(i,n)) + p.
\end{aligned}$$

Applying that to equations (13), (14), (15) produces

$$\begin{aligned}
&\hat{f}(k,(s(i)+p,p)) = p \\
&\hat{f}(k,(p,s(n)+p)) = n+1+k+p \\
&\hat{f}(k,(p,p)) = k+p
\end{aligned}$$

This equational definition is equivalent to the program:

$$\begin{aligned}
\hat{f}(k,(i,n)) = {} & \textbf{if } i > n \textbf{ then } n \\
& \textbf{else if } i = n \textbf{ then } n+k \\
& \textbf{else } n+k
\end{aligned}\,.$$

$$
\begin{array}{l}
\textbf{fun}\ \mathit{apply\_substitution\_}\sigma_0(M)\\
\quad =\\
\qquad \textbf{let}\ \ \textbf{fun}\\
\qquad\qquad \begin{array}{ll}
\widehat{Mwp}(k,(i,\mathit{Var}(n))) & = \textbf{if}\ i>n\ \textbf{then}\ \mathit{Var}(n)\ \textbf{else}\ \mathit{Var}(n+k)\\
\mid \widehat{Mwp}(k,(i,\mathit{Abs}(t))) & = \mathit{Abs}(\widehat{Mwp}(k,(s(i),t)))\\
\mid \widehat{Mwp}(k,(i,\mathit{App}(t,t'))) & = \mathit{App}(\widehat{Mwp}(k,(i,t)),\widehat{Mwp}(k,(i,t')))\\
\textbf{fun} & \\
\sigma(i,n) & = \textbf{if}\ i>n\ \textbf{then}\ \mathit{unit}(n)\\
 & \quad \textbf{else if}\ i=n\ \textbf{then}\ \widehat{Mwp}(i,(0,\mathit{Abs}(\mathit{App}(\mathit{Var}(0),\mathit{Var}(1)))))\\
 & \quad \textbf{else}\ \mathit{unit}(n-1)\\
\textbf{fun} & \\
\mathit{Ewp}(i,\mathit{Var}(n)) & = \sigma(i,n)\\
\mid \mathit{Ewp}(i,\mathit{Abs}(t)) & = \mathit{Abs}(\mathit{Ewp}(s(i),t))\\
\mid \mathit{Ewp}(i,\mathit{App}(t,t')) & = \mathit{App}(\mathit{Ewp}(i,t),\mathit{Ewp}(i,t'))
\end{array}\\
\qquad \textbf{in}\ \ \mathit{Ewp}(0,M))\\
\qquad \textbf{end}
\end{array}
$$

Figure 7: Final result

The program simplifies to:

$$\hat{f}(k,(i,n)) = \textbf{if}\ i > n\ \textbf{then}\ n\ \textbf{else}\ n+k$$

By unfolding $\hat{f}$ and by a well known property of **if** $\cdots$ **then** $\cdots$ **else** $\cdots$, equation (8) becomes:

$$\widehat{Mwp}(k,(i,\mathit{Var}(n))) = \textbf{if}\ i > n\ \textbf{then}\ \mathit{Var}(n)\ \textbf{else}\ \mathit{Var}(n+k)$$

Including the transformed form of σ, which comes from above, produces the program in Figure 7 which does not perform redundant computations for σ_i and f_i. The transformation involved in this section has been done manually. However the transformation process is systematic and involves equational reasoning using Theorem 1 and Theorem 2. It shows implicitly how to automatically transform a function of type $Nat * Nat \rightarrow Nat$ into a more efficient conditional form.

7 Directions

The paper has presented a clearly motivated and correct specification for a subtle representation of λ-terms, the implementation of which has, in the second authors experience, been prone to

"off by one errors." It has taken this abstract specification, with its extensive use of higher-order concepts, reduced it to a first-order program, introduced index arithmetic and produced an efficient algorithm that exploits computer arithmetic.

This development illustrates several new techniques. First, it makes the monadic structure in the development of the specification explicit by showing that it is a monad in FUNSEQ. It supports this structure with new program transformation techniques which allow the implicit use of arithmetic to be "rediscovered" formally. Finally, it demonstrates the feasibility of integrating tools for monadic programming and specification, which tend to be higher-order, with relatively standard program transformation technology, which is strictly first-order. The importance of partial evaluation technology in bridging this gap cannot be overstated.

7.1 Technology

Currently our technology is a tower of Babble. Automatic support for monadic programming, including automatic program generation, exists in TRPL, a language developed by Sheard[9, 5]. Hook uses Standard ML for examples of monadic program development because its module system is the most able to express the structure of the monads. The partial evaluator, Schism, uses its own (typed) dialect of Scheme as its object language. ASTRE, Bellegarde's program transformation system, is written in CAML. It uses a very simple first-order language as its object language.

In this environment, claims that the development is automatable mean that we have automated the process "piecewise", translating between the formalisms in a nearly mechanical fashion. It is, of course, our vision that one day these tools will all work in concert, allowing a development to proceed from specification to efficient realization with human intervention only when necessary.

7.2 Reuse

Although this paper has focused on the λ-calculus, the specification can be applied to virtually any abstract syntax with a regular binding structure provided its type can be expressed as a monad and the appropriate definition of *map_with_policy* can be given. For example, adding boolean constants and a conditional has no effect on the specification of substitution and only changes *map_with_policy* by defining it to apply f recursively on the components of the conditional without applying Z. Adding *let* is also trivial; again, no changes need to be made to the specification of substitution—only to *map_with_policy*. In this case, *map_with_policy* must apply Z to f when it enters the component in which the bound variable has been introduced. This ability to reuse specifications is one of the strongest arguments for the adoption of monads as a tool to structure program specification and development.

But what about the transformations? Can we reuse program improvements? Here we have less experience, however the decisions that are required to improve programs for the different scenarios outlined above are substantially the same. It appears that a transformation system that records its development may be able to replay the development and obtain similar improvements.

7.3 Application to more complex types

In earlier work more complicated types have been investigated. In particular, the following version of the *Term* type, which allows for a standard representation of variable names, was used:

$$\begin{array}{rcl} \textbf{datatype}\ \mathit{Term}(\alpha,\beta) & = & \mathit{Var}(\alpha) \\ & | & \mathit{Abs}(\beta,\ \mathit{Term}(\alpha)) \\ & | & \mathit{App}(\ \mathit{Term}(\alpha,\beta) * \mathit{Term}(\alpha,\beta)) \end{array}$$

In this representation $\lambda x\,.\,x$ would be represented

$$\mathit{Abs}(\mathbf{x}, \mathit{Var}(\mathbf{x}))$$

The techniques presented here all apply to this more general case, but the explanations are somewhat more involved. For example, instead of using a category where morphisms are sequences of functions indexed by numbers, a category of trees of functions indexed by sequences of elements from an arbitrary type must be used. Similarly, the data types introduced to encode the higher-order function applications become more complex.

References

[1] Françoise Bellegarde. Program transformation and rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 226–239, Berlin, 1991. Springer-Verlag.

[2] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972. Also appeared in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75(5).

[3] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. In *Proceedings of the Koninklijke Nederlandse Akaemie van Wetenschappen*, pages 348–356, Amsterdam, series A, volume 81(3), September 1978.

[4] Charles Consel. The Schism Manual, version 2.0. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1992.

[5] James Hook, Richard Kieburtz, and Tim Sheard. Generating programs by reflection. Technical Report 92-015, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1992.

[6] Richard B. Kieburtz. A generic specification of prettyprinters. Technical Report CSE-91-020, Department of Computer Science and Engineering, Oregon Graduate Institute, 1991.

[7] Eugenio Moggi. Notions of computations and monads, July 1991.

[8] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[9] Timothy Sheard. A user's guide to TRPL: A compile-time reflective programming language. Technical Report 90-109, Computer and Information Sciences, University of Massachusetts, Amherst, 1990.

[10] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1992.