Integrating Normalization-by-Evaluation into a Staged Programming Language

Tim Sheard
Pacific Software Research Center
Oregon Graduate Institute
sheard@cse.ogi.edu

1 Introduction
We have designed METAML, a multi-stage programming language, as a meta-programming system, i.e. a system which is used to write programs (meta-programs) whose sole purpose is to build and manipulate other programs (object-programs). METAML programs are simply ML programs which are annotated with staging operators.

There is a strong connection between off-line partial evaluation and staged programming. The staging operators of METAML can be thought of as a manual form of binding time analysis. In METAML, binding time annotations are first class operators, and users place them explicitly, rather than using an automatic binding time analysis to place them implicitly. In METAML, the reinitialization stage of an off-line partial evaluator is replaced by run-time code generation and compilation, and there is no upper limit on the number of stages.

While this is a powerful technique, there are, nevertheless, many simple programs where automatic binding time analysis is sufficient, and hand staging is simply an annoyance. In METAML we have combined the advantages of both, allowing a simple type-directed binding time analysis to co-exist with the manual staging annotations. This integration is the subject of this paper.

2 METAML, a multi-staged language
This section introduces METAML [9, 6], a statically-typed, multi-stage programming language. Here, we describe each of the staging operators of METAML.

2.1 The bracket operator: Building pieces of code
In METAML, a stage-1 expression is denoted by enclosing it between meta-brackets. For instance, the pieces of code denoting the constant 23 is given by:

```
|- <23>
val it = <23> : <int>
```

The expression `<23>` (pronounced “bracket 23”) has type `<int>` (pronounced “code of int”). Consider the following example where `length` refers to a previously defined function.

```
|- <length [1,2]>
val it = %<length [1,2]> : <int>
```

The `%` in the returned value indicates that `length` has been lifted from a value to a constant piece of code. We call this lexical capture of free variables. This is explained in more detail in section 2.5. Because in METAML operators (such as `+` and `*`) are also identifiers, free occurrences of operators often appear with `%` in front of them.

2.2 The escape operator: Composing pieces of code
Bracketed expressions can be viewed as frozen, i.e. evaluation does not apply under brackets. However, it is often convenient to allow some reduction steps inside a large frozen expression while it is being constructed. METAML allows one to escape from a frozen expression by prefixing a sub-expression within it with a tilde (\`). Because tilde must only appear inside brackets, it can only be used at level-1 and higher. For instance, let us examine the function `pair` below:

```
|- fun pair x = (`x, `x)
val pair = fn : ['b], 'b->-> (<'b * 'b>)
```

The function `pair` takes a piece of code (of type `<'b>`) as input, and produces a new piece of code (of type `<('b * 'b)>`). It transforms the input code `x` into the code of the pair `(x, x)`. To do this we must “splice” `x` into the resulting code in two places. This is done by escaping the occurrences of `x` in the definition of pair.

When `v` appears inside brackets at level-1, the system evaluates `v` to a piece of code `<v>`. Then `v` is spliced into the bracketed expression at the location where the escaped expression lexically appears.

The function `pair` can be used to construct new code from old:

```
|- (pair <17-4>)
val it = ((17 % 4, 17 % 4)) : (<int * int>)
```

2.3 The run operator: Executing user constructed code
The `run` operator is the explicit annotation used to indicate that it is now time to execute a delayed computation (i.e. a piece of code).
- | val z = 27 - 15;
  val z = 27 \% 15 : int

- | run z;
  val it = 12 : int

The run operator allows us to reduce a piece of code to a value by executing the code. Computation is no longer deferred and the resulting value is a pure value.

2.4 The lift operator: Another way to build code

Similar to meta-brackets, lift transforms an expression into a piece of code. But lift differs in that it reduces its input before freezing it. This is contrasted in the examples below.

- | <4+1>;
  val it = <4 \% 1> : <int> (* no execution *)

- | lift (4+1);
  val it = 5 : <int> (* 4+1 executed *)

It should also be noted that lift cannot be applied to a higher-order (i.e. functional) arguments, as it is undefined on them.

2.5 Lexical capture of free variables: Constant pieces of code

As mentioned earlier any value can be lifted into a constant in a later stage. We call this cross-stage persistence. We illustrate the differences between the different ways to construct pieces of code in the following:

- | val z = 3+4;
  val z = 7 : int

- | val quad =
  (3+4, lift (3+4), <2>);

- | val quad = (7, lift (3 \% 4), <7>, lift (3 \% 2));
  (int * lift (int) * lift (int))

Here, the variable z is defined at stage 0, but inside the fourth component of the tuple quad (where it occurs free), it is referenced at stage 1. At runtime, when the expression <2> is evaluated, the system has to compute a piece of code related to the value of z. This piece of code will be a constant, because z is known to be 7. We call this phenomenon cross-stage persistence[9]. The pretty printer for code prints all lexically captured constants with the annotation \%, followed by the name of the free variable whose value was used to construct the constant. All free variables (regardless of type) inside meta-brackets construct these constants. This is the way functions are made into code.

3 Normalization by evaluation

Normalization by Evaluation (NBE) (or type-directed partial evaluation) can be thought of as a simple binding time analysis which uses type information to place binding time annotations. In a language where staging annotations are first class, this can be quite convenient. We have implemented several versions NBE in META ML as primitives, and built type-safe interfaces to them for the user.

Staged programming requires placing staging annotations in a program to take advantage of run-time invariants. For example a staged interpreter takes advantage of the run-time invariant that the program it is interpreting is a constant. It does this by generating a specialized program in the first stage, and executing that program in the second stage. Thus, it behaves like a compiler. How the annotations are place is irrelevant. In META ML annotations are placed either manually, or by using an automatic staging mechanism based upon NBE.

The interface to NBE is the predefined polymorphic function reify. It has type: 'a -> '<a> where 'a is a universally quantified type variable. It is implemented as follows: the META ML Hindley-Milner type inference mechanism, translates META ML programs into the polymorphic lambda calculus. META ML is implemented as an interpreter for the polymorphic lambda calculus. Thus at run-time, when reify is applied, the system knows to what type 'a is bound. The implementation of reify does an intensional analysis of 'a to place binding time annotations. See Section 4 for details.

This feature is generally used by writing an unstaged version of a function. Partially applying the function to some of its arguments, and reifying the result. The code produced is then incorporated into the manual staging process using the escape operator.

To illustrate, suppose a function f is needed which tests, for two integers m and n, if m is greater than 0, even, and also less than n. Suppose n is known in an early stage, before m becomes known, and a version of this function specialized to n is to be generated in the early stage. One can generate this specialized version using reify and library functions, no hand annotation necessary for this part of the process.

fun member [] x = false
  | member (y::ys) x =
    if x=y then true else member ys x;

fun count m n =
  if m >'n then []
  else m::(count (m+1) n);

fun filter p [] = []
  | filter p (x::xs) =
    if p x then x ::(filter p xs)
    else filter p xs;

fun even x = (x mod 2)=0;

fun f m n =
  member (filter even (count 1 n) m);

By partially applying f to n a function of type int -> bool is obtained. Reifying this we obtain a piece of code. For example reify (f 5) returns the following piece of code:

<fn a -> if a \% 2
  then true
  else if a \% 4
    then true
    else false>

This code is then incorporated into the larger second stage code.
3.1 Dangers of NBE

NBE is a very useful, but not very sophisticated process. It is easy to build values for which the reification function does terminate. The termination of the above example is guided strictly by the static value \( n \). If termination depended upon the value of \( m \), reification would not terminate.

The strength of the integrated approach, using both manual annotations and automatic annotation using NBE, is that the user can control non-termination behaviors explicitly.

3.2 Extended Example

We illustrate this by an extended example: providing polytypic functions. A polytypic function is defined just once, but usable on many different datatypes. Generalized map functions, and structural equality are examples.

In METAML, we proceed as follows. Define a universal domain as a datatype. Every first-order datatype with a single type parameter (such as list) can be mapped into this universal domain. When this is done the datatype looses its "type". We encode the generic function as a function over values in the universal domain (thus applicable to all such datatypes). After applying the generic function, the answer is projected from the universal domain back into the typed domain. The strategy uses staging to specialize the generic function and thus remove all reference to the universal domain. We illustrate for first-order datatypes with a single parameter. Call these datatypes \( R \). The universal domain is encoded in the datatype \( V \) (for variant below), list two parameters encode the type parameter \( \gamma \), and recursive component(\( s \)) \( \gamma \) of the set of datatypes \( R \).

datatype \((\gamma \gamma) \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \gamma \ga
While elegant, it is not very efficient. The rolling and unrolling consumes a lot of resources (allocation of memory for the constructors of the V datatype), and time (spent walking over the V data structure). Given a particular T reg structure we would like to partially apply mp and then partially evaluate away these inefficiencies. Unfortunately reification of mp list does not terminate, because termination is controlled by the list which is unknown. The combination of manual staging with automatic staging comes to the rescue.

Write a helper function that combines the unrolling, the pushing of the mapping function and the recursive call, and the rolling-up, but not the recursion.

```haskell
fun q r f mapf x = 
    rollup r (V f mapf (unroll r x));
```

```haskell
fun make r =
    <fn f =>
    let fun map x = `'(reify(q r)) f map x in map end>

Stage the recursive call, reify the helper function and splice it into the staged recursive call.

When applied to T reg structure the function make will construct a specialized version of the map function. For example:

```haskell
-| make list;
val it =
<fn a => let fun b c =
   (case c of [] => []
       | e::d => :>(a e, b d))
in b end>
```

We have used the same strategy to construct many polytypic functions such as show, equality, fold and crush.

4 Implementing NBE in a staged language

Because METAML has binding time annotations as first class features, it is almost possible to express NBE as a source level program. Almost for several reasons. First, in a Hindley-Milner typed language, one does not compute over types in the manner that NBE does. Second, because the program is not well typed under the typing rules of METAML. Nevertheless, we have implemented several versions of NBE underneath the type-checker, as primitives functions, and built type-safe interfaces to them for the user.

In [1] NBE is expressed as two mutually recursive functions reify and reflect. In Figure 3.2 we give continuation passing style versions of reify and reflect as two stage programs. This allows an implementation without the abstract control operators shift and reset [2, 4].

We introduce a datatype T which can be thought of as a representation of types, and reify and reflect functions whose control is guided by their T typed arguments.

The function reify has type T -> 'a -> ("an" -> "an") -> 'a. Its operation can be described as follows: given a T value typ representing a type, and value v with the type represented by typ, produce a piece of code representing v. Finally, apply the code-to-code continuation k to produce the answer (also a piece of code). When the continuation is the identity, this function behaves like the reify function of [1].

The function reflect has type T -> "an" -> ("an" -> "an") -> "an". It's purpose is to construct the body of the lambda expression being constructed in the function clause of reify. Given a T value typ representing a type (the domain of the lambda being constructed), and a piece of code e (the variable of the lambda), with the type represented by typ, produce a piece of code representing the body of the lambda. It does this by combining two things. First, the code valued argument e, and second, the code obtained by applying the value-to-code continuation k. It is the job of reify to construct an expanded value (with the "shape" encoded in its T valued argument typ) and then apply the continuation k to obtain this second piece of code.

4.1 Generalized expansion

The purpose of generalized expansion in reflect is to make the structure of a value's type explicit, so that reflect's continuation, which "needs" a value of that shape, will succeed. Note how the continuation is always applied to an explicit expression with the "shape" corresponding to the type. We illustrate this expansion with the following table:

<table>
<thead>
<tr>
<th>type</th>
<th>expansion code</th>
</tr>
</thead>
<tbody>
<tr>
<td>f : 'a -&gt; 'a</td>
<td>k(fn x =&gt; f x)</td>
</tr>
<tr>
<td>p : 'a * 'a</td>
<td>case p of (a,b) =&gt; k(a,b)</td>
</tr>
<tr>
<td>s : 'a + 'a</td>
<td>case s of Left a =&gt; k(Left a)</td>
</tr>
<tr>
<td></td>
<td>Right b =&gt; k(Right b)</td>
</tr>
</tbody>
</table>

The function reflect implements expansion as well as handling the staging annotations. Note that at a function type, generalized expansion is simply eta-expansion[3].

4.2 Things to Note about Figure 3.2

Note that reify never does any analysis of its argument v. The context, provided by the analysis of the argument typ always "knows" what type v has. Reify uses v in its body at several different types. This is why reify is typed. Note that the lift operator must also be parameterized by the type (lift[typ]) as well to maintain the non-analysis invariant.

Doing no analysis over v is important because it places few restrictions on an implementation. In particular NBE could be implemented in a compiled language such as Scheme or ML[1].

4.3 Limitations of Figure 3.2

Our experience has been that reify and reflect as defined in Figure 3.2 work only for values derived from closed terms where base types (int, string or real for example) only appear in covariant positions in the type represented by typ. And only in covariant positions because of the primitive lift operator.

We conjecture that one could formalize this using a parametricity argument, using logical relations. We are currently working on this.
datatype T = Alpha | Arrow of T*T | Prod of T*T | Sum of T*T | Base;

(* reify : T -> 'a -> (<'a'> -> 'a') -> 'a > *)

fun reify typ v k =
case typ of
  Alpha => k(valueToCode v)
| Base => k(lift[typ] v)
| Arrow(d,r) => k <fn x => \(\text{reflect } d \times x\) (fn z => reify r (v z) id))>
| Prod(x,y) => \(\text{case } v \text{ of } (a,b) \Rightarrow \text{reify } x \ a \ (fn \ e1 \Rightarrow \text{reify } y \ b \ (fn \ e2 \Rightarrow k \langle \sim \ e1, \sim e2 \rangle))\)
| Sum(x,y) => \(\text{case } v \text{ of Left } a \Rightarrow \text{reify } x \ a \ (fn \ e \Rightarrow k \langle \text{Left } e \rangle))
| Right b => \(\text{reify } y \ b \ (fn \ e \Rightarrow k \langle \text{Right } e \rangle))\)

(* reflect : T -> 'a > ('a -> 'a) -> 'a > *)

and reflect typ e k =
case typ of
  Alpha => k(codeToValue e)
| Base => Error "Base type in contravariant position"
| Arrow(d,r) => k <fn v => run(reflect v (fn arg => reflect r \<-' \arg > (fn \ x => \<x>)))\)
| Prod(x,y) => \(\text{case } e \text{ of } (a,b) \Rightarrow \text{reflect } x \ a \ (fn \ v1 \Rightarrow \text{reflect } y \ b \ (fn \ v2 \Rightarrow k \langle v1,v2 \rangle))\)
| Sum(x,y) => \(\text{case } e \text{ of Left } a \Rightarrow \text{reflect } x \ a \ (fn \ \ m \Rightarrow k \langle \text{Left } \ m \rangle))
| Right b => \(\text{reflect } y \ b \ (fn \ \ m \Rightarrow k \langle \text{Right } \ m \rangle))\)

Figure 1: Continuation passing style reify and reflect as two stage programs. The primitive functions:
valueToCode : 'a -> 'a' and codeToValue : 'a' -> 'a' are simply coercion functions, and could be implemented by the identity.

Base types in contravariant positions seem problematic. Consider the ML function: (fn 5 => 0 | fn _ => 1) with type int -> int. It is hard to reify such a value without doing an intensional analysis of the term from which it was derived. One could argue that the problem in this term stems from pattern matching over constants (which requires equality), so this term isn't really closed.

But consider the function: (fn f => fn y => if true then f y else 0). This function has type: ("a -> int -> int") -> 'a -> int. Note that the second occurrence of int is in a contravariant position in the whole term.

Stepping through evaluation of reify and reflect, the problem arises when one attempts to lift a value of type int which is not an integer, but a piece of code representing the application f y 1. This can be fixed by allowing lift to analyze its argument, playing the role of the identity if it is passed a piece of code. This allows a stronger system, but significantly restricts the implementation, requiring tagging on values to test if they are piece of code or not.

In METAML, which is implemented as an interpreter in ML, we already have type tags on values. We have implemented both strategies. The second strategy can be strengthened to handle primitive operators, and open terms, but is considerably more complicated [8].

References