Using MetaML: a Staged Programming Language

Tim Sheard

Oregon Graduate Institute

1 Why Staging?

The purpose of staged programming in general, and MetaML in particular, is to produce efficient programs. We wish to move beyond programs that compute the "correct" output, to those that also have better control over resources (both space and time). The mechanism for doing this is to use program annotations to control the order of evaluation of terms. It should come as no surprise to those who have studied the λ-calculus that the number of steps in a reduction is strongly influenced by the order of evaluation. Since the number of steps in a reduction relates strongly to the resources it consumes, controlling evaluation order gives programmers better control over resources consumed. MetaML allows programmers to move beyond a fixed evaluation strategy, and to specify precisely the desired evaluation order.

This provides a mechanism which allows general purpose programs (written in an interpretive style that eases both maintenance and construction) to perform without the interpretive overhead usually associated with such programs.

Much of the rivalry between lazy functional languages (such as Haskell) and strict functional languages (such as Standard ML) comes from the perceived superiority of one fixed evaluation order (outermost for lazy, innermost for strict) over another. But this perceived superiority is just that, *perceived*. Recent work, especially that of Chris Okasaki [8] on functional data structures, has shown that no single fixed evaluation order is superior in all cases.

There have been attempts at controlling evaluation order in the past. Strictness annotations in lazy languages temporarily employ an eager evaluation strategy, and constructs such as *force* and *delay* employ a lazy strategy in an strict language. It is also possible (in an strict language) to simulate laziness by using the delaying effect of lambda abstraction. For example, a typical simulation of lazy lists in a strict language might be defined as:

```
datatype 'a lazylist =
            lazyNil | lazyCons of 'a * (unit -> 'a lazylist);

fun count n = lazyCons(n, fn () => count (n+1))
```

Where the tail of a list is a function that forces the computation of the next element, but only when applied.

The lambda expression, the basis of a language with first class functions, is both a blessing and a curse. It is a blessing since it allows us to build abstractions, which can be used many times. As illustrated, it allows programmers to construct lazy, infinite data structures even in a strict language, but it is a curse because it never allows computation under lambda until the lambda is applied. This profound limitation applies equally to lazy and strict languages. Sometimes computation under the lambda is exactly what is called for, yet we have no way of expressing this. I will try and illustrate this point below:

```
fun power n = (fn x => if n=0 then 1 else x * (power (n-1) x))

map (power 2) [1,2,3,4,5]
```

This defines a generic `power` function, and a small program where the `power` function is specialized to the square function (the exponent `n` is fixed at 2), and then this specialization is repeatedly applied many times by the `map` function.

The most efficient strategy is to unfold the definition of `power` once, but since the result of unfolding `power 2` is a lambda no computation is really performed. Suppose we could direct which reductions were to be done, even under lambda. Then, by using the comments to choose which reduction step to employ, we proceed as follows

```
map (power 2) [1,2,3,4,5]
       (* unfold the definition *)
map (fn x => if 2=0 then 1 else x * (power (2-1) x)) [1,2,3,4,5]
       (* perform the if, under the lambda *)
map (fn x => x * (power (2-1) x)) [1,2,3,4,5]
       (* unfold power again *)
map (fn x => x * ((fn x => if 1=0 then 1 else x * (power (1-1) x)) x))
    [1,2,3,4,5]
       (* use the beta rule to apply the explicit lambda to x *)
map (fn x => x * (if 1=0 then 1 else x * (power (1-1) x))) [1,2,3,4,5]
       (* perform the if *)
map (fn x => x * (x * (power (1-1) x))) [1,2,3,4,5]
       (* unfold power again *)
map (fn x => x * (x * (fn x => if 0=0 then 1 else x * (power (0-1) x)) x))
    [1,2,3,4,5]
       (* use the beta rule to apply the explicit lambda to x *)
map (fn x => x * (x * (if 0=0 then 1 else x * (power (0-1) x))))
    [1,2,3,4,5]
       (* perform the if *)
map (fn x => x * (x * 1)) [1,2,3,4,5]
       (* apply the map *)
[1,4,9,16,25]
```

Only after completely unfolding the `power` function do we use the `map` function to repeatedly apply the squaring function. We could only unfold `power` by applying reduction rules under lambda. This saves the duplicated reductions which would unfold the power function each time `power 2` is applied.

In MetaML we annotate a program to provide exactly this kind of knowledge. We use three annotations. One to delay the reduction of an expression, one to splice two delayed expressions together to build a larger delayed expression, and one to force a delayed expression to be reduced.

- We use brackets (`< >`) to surround expressions to indicate that reduction should *not* occur on the expression inside the brackets. We call such a delayed expression a piece of code. Brackets are the introduction rule for code.
- Inside brackets we use escape (`~`) to relax the restriction that no reductions may occur brackets inside brackets. That is, only escaped expressions may be reduced inside brackets. This provides a mechanism to splice two pieces of code together to form a larger piece. Inside brackets, if an escaped expression reduces to a bracketed one, then both the escape and the inner bracket my be removed (e.g. `< ... ~ <e> ...>` reduces to `<... e ...>`). This is the first elimination rule for code.
- Finally, we use `run` to remove outermost brackets. This forces a piece of code to be evaluated (`run < e >` reduces to `e`). This rule only applies when no escapes remain in the bracketed expression `e`. This is the second elimination rule for code

It is the first elimination rule for code (escape bracket cancellation) that makes MetaML so expressive. The escaped expression can be anywhere, even under a lambda! E.g. `<fn x => ... ~e ...>` forces evaluation of `e` even though it would ordinarily be delayed until the lambda was applied. This is a very powerful and expressive construct as we shall see.

After all brackets have been eliminated the default evaluation strategy applies to the remaining term. With these annotations we rewrite the power example as follows:

```
fun power n =  fn x => if n=0
                          then <1>
                          else < ~x * ~(power (n-1) x) >)

map (run <fn z => ~(power 2 <z>)>) [1,2,3,4,5]
```

This is just an annotated version of the original example (except that context requires the call to `(power 2)` to be eta-expanded).

Using (the default) strict[1] evaluation strategy, but following the order implied by the annotations (rather than using explicit comments to direct evaluation under lambda as in the previous example), we proceed:

```
map (run <fn z => ~(power 2 <z>)>) [1,2,3,4,5]

map (run <fn z => ~(if 2=0 then <1> else < ~<z> * ~(power (2-1) <z>) >)>)
    [1,2,3,4,5]
```

[1] by strict we mean a leftmost, innermost strategy

```
map (run <fn z => ~< ~<z> * ~(power (2-1) <z>) >>) [1,2,3,4,5]

map (run <fn z => ~< z * ~(power (2-1) <z>) >>) [1,2,3,4,5]

map (run <fn z => ~< z * ~(if 1=0 then <1>
                               else < ~<z> * ~(power (1-1) <z>) >) >>)
    [1,2,3,4,5]

map (run <fn z => ~< z * ~< ~<z> * ~(power (1-1) <z>) >>>) [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * ~(power (1-1) <z>) >>>) [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * ~(power 0 <z>) >>>) [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * ~<1> >>>) [1,2,3,4,5]

map (run <fn z => ~< z * ~< z * 1 >>>) [1,2,3,4,5]

map (run <fn z => ~< z * z * 1 >>) [1,2,3,4,5]

map (run <fn z => z * z * 1 >) [1,2,3,4,5]

map (fn z => z * z * 1) [1,2,3,4,5]

[1,4,9,16,25]
```

This simple idea is the key to staged programming. It can have a profound effect on the way programs are constructed and used. The ability to direct reduction under a lambda makes this paradigm strictly more powerful than traditional paradigms with a single fixed evaluation strategy. The staged paradigm does not allow more programs to be expressed, but instead allows all programs to control their own evaluation order, and thus have more control over their resource consumption.

2 Relationship to other paradigms

MetaML is strongly related to several other programming paradigms. In particular lisp-like macros with eval, meta-programming, program generation, and partial evaluation.

Lisp-like macros. MetaML's three annotations, bracket, escape and run, are analagous to Lisp's back-quote, comma and eval. Brackets are similar to back-quote. Escape is similar to comma. Run is similar to eval in the empty environment. However, the analogy is not perfect. Lisp does not ensure that variables (atoms) occurring in a back-quoted expression are bound according to the rules of static scoping. For example `` `(plus 3 5) `` does not bind `plus` in the scope where the back-quoted term appears. This is an important feature of

MetaML. In addition, **Lisp** employs a dynamic typing discipline, while MetaML employs a static typing discipline, an important distinction.

Meta-programming. In MetaML a bracketed expression is considered a piece of code. We think of code as a data structure which can be manipulated like any other, but with the additional ability that it can be `run`.

Because a MetaML program manipulates programs (represented by code) MetaML is a meta-programming system. In MetaML both the meta-language (the language that describes the manipulations) and the object language (the language of the programs being manipulated) are the same: ML.

Program generation. One solution to inefficient interpretive programs is to write a program generator. Rather than write a general purpose but inefficient program, one writes a program generator that generates an efficient solution from a specification. This provides a natural staging to the solution. The use of the parser generator Yacc is an illustrative example. Rather than using a general purpose parsing program, we generate an efficient parser from a specification, i.e. a language grammar. MetaML provides a uniform environment for constructing program generators in a single paradigm. It provides an approach radically different from, and superior to, the ad-hoc "programs-as-strings" view that seems to predominate in most software generation systems.

Partial evaluation. Partial evaluation optimizes a program using a-priori information about some of that program's inputs. The goal is to identify and perform as many computations as possible in a program before run-time.

The most common type of partial evaluation, *Off-line* partial evaluation, has two distinct steps, *binding-time analysis* (BTA) and *specialization*. BTA determines which computations can be performed in an earlier stage given the names of inputs available before run-time (static inputs).

In essence, BTA performs automatic staging of the input program. After BTA, the actual values of the inputs are made available to the specializer. Following the annotations, the specializer either performs a computation, or produces text for inclusion in the output (*residual*) program.

The relationship between partial-evaluation and staged programming is that the intermediate data structure between the two steps is a *two-stage annotated program* [1], and that the specialization phase is the execution of the first stage in the two-stage annotated program produced by BTA.

3 Introducing MetaML

This section provides a gentle introduction to MetaML [12, 6]. We designed MetaML 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 provides built-in support for a number of hard problems that repeatedly occur in meta-programming and generation systems. This support includes:

- A type system ensuring the well-formedness (type-safety) of object programs from a type analysis of the meta-program which produces them. This is crucial when debugging multi-stage programs because it reports type errors in the object-programs at the compile-time of the meta-programs, not when the object-programs are executed.
- The capability to use arbitrary values from the meta-program as constants in the generated program. This provides a solution to the *hygienic macro* problem in a typed language, i.e. it supports macro-like constructs which bind identifiers in the environment of definition, not in the environment of their expansion. This completely eliminates inadvertent "capture" problems, and is an implementation of static scoping in a staged language.
- The capability to display code. When debugging, it is important for users to observe the code produced by their programs. This implies a display mechanism (pretty-printer) for values of type code.
- The capability to perform "generation-time" optimization on generated code. Generated code is a first class piece of data and can be manipulated to effect optimizations etc.
- The capability to "execute" the code built for testing and prototyping purposes.

MetaML programs are simply ML programs which are annotated with staging operators. In the next section we describe each of the staging operators and introduce the MetaML language by using short, self-contained "sessions" of the actual implementation. MetaML uses a *read-typecheck-eval-print* top level loop. An expression is entered after the prompt (`|-`), it is type checked, evaluated, and then its name, value and type are printed.

3.1 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 illustrated by the following MetaML session:

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

The expression `<23>` (pronounced "bracket 23") has type `<int>` (pronounced "code of int"). The types `int` and `<int>` are not the same. Trying to use `<23>` as an integer fails in the type checking stage, and the system complains:

```
-| <23> + 2;
Type Error:
  Cannot unify the types: in type application the type constructors
  do not match: int is not equal to <int>
 in expression: (<23>,2)
```

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 or *cross stage persistence*. This is explained in more detail in section 3.5. Because in MetaML operators (such as `+` and `*`) are also identifiers, free occurrences of operators often appear with `%` in front of them when code is displayed.

Bracketed lambdas. Any expression can be delayed, including higher order (functional) expressions. Consider the examples:

```
-| val idCode = <fn x => x>;
val idCode = <fn a => a> : ['b].<'b -> 'b>

-| <fn n => n + 1>;
val it = <fn a => a %+ 1> : <int -> int>
```

`idCode` is the code representing a pure MetaML function, the type associated to `idCode` is `['b].<'b -> 'b>` which is a polymorphic piece of code with polymorphic[2] variable `'b`. Note that the display mechanism for code alpha-renames bound variables hence the `(fn a => a)`. `<fn n => n + 1>` denotes the representation of a function. It is the encoding of the increment function over integers.

The *level* of any piece of code is the number of surrounding brackets minus the number of surrounding escapes. Simple values such as `13` and `(fn n => n+1)` are level-0 code. In `<fn n => n + 1>`, the function inside the brackets is level-1 code. Finally, in term `<fn n => <n + 1>>`, the sub-term `n + 1` is a level-2 piece of code.

```
-| <fn n => <n + 1>>;
val it = <fn a => <a %+ 1>> : <int -> <int>>
```

`<fn n => <n + 1>>` denotes a three stage program. In stage-0 it is simply a piece of data which represents a program (`<fn n => <n+1>>`). The result of running that program is a function that can be used in stage 1 (`fn n => <n+1>`). When applied to an integer that function produces another piece of code, which can be used in stage 2 (`<%n %+ 1>`).

3.2 The escape operator: composing pieces of code

Bracketed expressions can be viewed as *delayed*, i.e. evaluation does not apply under brackets. However, it is often convenient to allow some reduction steps inside a large delayed expression while it is being constructed. MetaML allows one to *escape* from a delayed 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:

[2] The treatment of polymorphism in MetaML is actually quite subtle, the full treatment of polymorphism is beyond the scope of this paper. See [6] for more details

```
-| 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 `~ e` appears inside brackets at level 1, the system evaluates `e` to a piece of code `<v>`. Then `v` is spliced into the bracketed expression in the context where the original escaped expression occurred. This is the first elimination rule for code. It is an elimination rule since it shows how escape removes brackets.

The purpose of escape is to construct larger pieces of code by splicing smaller pieces of code together. Consider the function `pair`, which is used to construct new code from old:

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

By using the first elimination rule for code (`~<e>` rewrites to `e`), this reduction proceeds as follows:

```
pair <17 %- 4>

<( ~<17 %- 4>, ~<17 %- 4> )>

<( 17 %- 4, 17 %- 4 )>
```

3.3 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. The second elimination rule for code (`run <e>` rewrites to `e`) can *only* be applied if `e` does not contain escaped expressions. If `e` does contain escaped expressions, they must be evaluated and then eliminated using the first elimination rule for code before the run elimination rule applies. This is an important rule, since it forces a piece of code to be "fully expanded" before it can be run.

```
run <1 + ~( (fn x => x) <2+3> )>
```

```
run <1 + ~( <2+3> ) >

run <1 + ~<2+3> >

run <1 + 2+3 >

1 + 2+3

6
```

N-stage code is executed by N applications of the `run` annotation.

```
-| val x = <fn n => <n + 1> >;
val x = <fn a => <a %+ 1> > : <int -> <int>>

-| val y = run x;
val y = fn : int -> <int>

-| val z = y 6;
val z = <%n %+ 1> : <int>

-| run z
val it = 7 : int
```

3.4 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 delaying it. This is contrasted in the examples below.

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

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

Lift can be used to make the 2 stage example of the previous section more comprehensible. By using lift in the second stage the bound variable `n` appears as a literal constant (`6` below) rather than a lexically captured constant (`%n` in the previous example).

```
-| val x = <fn n => < ~(lift n) + 1> >;
val x = <fn a => <~(lift a) %+ 1> > : <int -> <int>>

-| val y = run x;
val y = fn : int -> <int>

-| val z = y 6;
val z = <6 %+ 1> : <int>
```

```
-| run z
val it = 7 : int
```

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

3.5 Lexical capture of free variables: constant pieces of code

As illustrated in the two stage example, it is often useful to construct code containing variables referring to values previously defined in an earlier stage. For example:

```
-| val n = 10;
val n = 10 : int

-| val codePair = <(n,3)>;
val codePair = <(%n,3)> : <(int * int)>
```

Here, the variable `n` is defined at stage 0, but inside `codePair` (where it occurs free), it is referenced at stage 1. At runtime, when the expression `<(n,3)>` is evaluated, the system has to compute a piece of code related to the *value of* `n`. This piece of code will be a constant, because `n` is known to be `10`. We call this phenomenon *cross stage persistence*[12]. 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.

Differences between lift and lexical capture. The `lift` operator cannot be used on functional values. This is because lift must construct an expression, which when evaluated returns the same result. For functions this is not always possible. With cross stage persistence we *can* lift a function into a piece of code. Cross stage persistence constructs a constant, which needs no evaluation when it is finally run. This allows us to construct code for functions.

```
-| val inc = fn a => a+1;
val inc = fn : int -> int

-| val encodeInc = <inc 5>;
val encodeInc = <%inc 5> : <int>

-| run encodeInc;
val it = 6 : int
```

We use `lift` when we want value in a previous stage to appear as a literal constant in the code representing a future stage.

4 Pattern matching against code

Since code is just a data structure it is possible to pattern match against pieces of code. Code patterns are constructed by placing brackets around code. For example a pattern that matches the litteral `5` can be constructed by:

```
-| fun is5 <5> = true
     | is5 _ = false;
val is5 = fn  : <int> -> bool

-| is5 (lift (1+4));
val it = true  : bool

-| is5 <0>;
val it = false  : bool
```

The function `is5` matches its argument to the constant pattern `<5>` if it succeeds it returns `true` else `false`.

Pattern variables in code patterns are indicated by escaping variables in the code pattern.

```
-| fun parts < ~x + ~y > = SOME(x,y)
     | parts _ = NONE;
val parts = fn  : <int> -> (<int> * <int>) option

-| parts <6 + 7>;
val it = SOME (<6>,<7>) : (<int> * <int>) option

-| parts <2>;
val it = NONE  : (<int> * <int>) option
```

The function `parts` matches its argument against the pattern `< ~ x + ~ y >`. If its argument is a piece of code which the is the addition of two sub terms, it binds the pattern variable `x` to the left subterm and the pattern variable `y` to the right subterm.

Code patterns which contain pieces of code with binding occurrences must use higher-order pattern variables. A higher-order pattern variable is indicated by an escaped application. This application must have a special form. It must be the application of a variable to arguments. This introduces a higher-order pattern variable. The arguments of the variable must be explicit bracketed variables, one for each variable bound in the code pattern at the context where the escaped application appears. For example consider the following patterns:

`<fn x => ~ (f <x>)>`	legal
`<fn x => ~ (f <2>)`	illegal, `<2>` is not a bracketed variable
`<fn x => ~ f>`	illegal, `f`, under lambda, is not applied to an argument
`<fn x => fn y => ~ (f <x>)>`	illegal, `f` not applied to all bound variables
`<fn (x,y) => ~ (f <x> <y>) + 1>`	legal

A higher order pattern variable is used like a function on the right hand side of a matching construct. For example a function which applies the rule that `0` is the identity of addition to the body of function is written as:

```
-| fun f <fn x => ~(g <x>) + 0> = <fn y => ~(g <y>)>
     | f x = x;
val f = Fn  : ['b].<'b  -> int> -> <'b  -> int>

-| f <fn x => (x-4) + 0>;
val it = <(fn a => a %- 4)> : <int -> int>
```

In the next sections we give several substantial examples which illustrate program staging.

5 A staged term rewriting system

One may think of a *term-rewriting system* as a set of directed rules. Each rule is made of a left-hand side and a right-hand side. Both the left-hand side and right-hand side of a rule are made of patterns. A pattern is a term with pattern matching variables as subterms.

A rule may be applied to a term t if a subterm s of t matches the left-hand side under some substitution σ. A rule is applied by replacing s with t', where t' is the result of applying the substitution σ to the right-hand side. We say *"t rewrites (in one step) to t'"*, and write $t \Rightarrow t'$. As an example, here are the rules for a Monoid [2]:

$$\begin{aligned} r_1 :& \quad x + 0 \rightarrow x \\ r_2 :& \quad 0 + x \rightarrow x \\ r_3 :& \; x + (y + z) \rightarrow (x + y) + z \end{aligned}$$

Variables x, y, and z in the rules can each match any term. If a variable occurs more than once on the left-hand side of a rule, all occurrences must match identical terms.

Generally, the rules do not change over the life of the system. At the same time, the basic form of the matching function is a simultaneous traversal of a subject term and the left-hand side of the rule it is being matched against. This offers an opportunity for staging: We can "specialize" matching over the rules in a first stage, and eliminate the overhead of traversing the left-hand side of the rules. Not only that, but as we will see, we can also remove a significant amount of administrative computations involved in constructing and applying the substitution σ. One would expect that this would significantly speed up the rewriting system.

In our system we have both patterns (with variables) and terms (without variables) We capture this with the following data structures:

```
datatype 'a Structure =
  Op of ('a * string * 'a)   (* e.g. (1 + 5) *)
| Int of int;                (* e.g. 5       *)

datatype term = Wrap of term Structure;

datatype pat =
  Var of string
| Term of pat Structure;
```

In the following algorithm it is necessary to compare two terms for equality if a pattern variable occurs more than once in the same pattern. Such a function is easy to write as a simultaneous traversal over two terms.

```
fun termeq (Wrap t1) (Wrap t2) =
case t1 of
  Op(m,s,n) =>
   (case t2 of Op(a,b,c) =>
      if s=b
         then (if termeq m a
                 then termeq n c
                 else false)
         else false
    | _ => false)
| Int n =>
   (case t2 of
      Int m => n=m
    | _ => false);
```

Because we are constructing a staged version of a pattern matcher, it is necessary to define a staged version of the substitution function. The `subst` function takes a substitution (binding variable names to code of terms) a pattern (containing variables) and produces the code of a term (without any variables). Since substitutions are implemented as lists, we need a auxillary function for looking things up in lists. The difference between this function and normal subsitution is that `subst` manipulates pieces of code (which will compute a term) rather than terms themselves.

```
fun find s [] = NONE
  | find s ((a,z)::xs) =
       if a=s then SOME z else find s xs

(* subst: (string * <term>) list  -> pat  -> <term> *)
fun subst sigma pat =
case pat of
  Var v =>
    (case find v sigma of
       SOME w => w)
| Term(Int i) => <Wrap(Int ~(lift i))>
```

```
| Term(Op(t1,s,t2)) =>
     <Wrap(Op (~(subst sigma t1),
               ~(lift s),
               ~(subst sigma t2)))>
```

Note the use of staging annotations to construct the code corresponding to the pattern.

A staged matcher takes a pattern and produces the code which matches a term against that pattern. The pattern is completely known, and the code produced depends upon the pattern. In the rewrite system the code we want to produce will build an instance of the right-hand side of a rule if the left-hand side matches the term. The instance depends upon the substitution built by the matching. Rather than returning a substitution (which is then applied to the right-hand side) the matching function is given a continuation which it should apply to the substitution. It is the continuation that builds the instance of the right-hand side, not the matching function.

```
type substitution = ((string * <term>) list) option;
type continuation = substitution -> <term>;

fun unWrap (Wrap x) = x;
fun unWrapCode <Wrap ~t> = t
  | unWrapCode e = <unWrap ~e>;

(* match : pat -> continuation -> substitution -> <term> -> <term>          *)
fun match pat       (* the pattern being matched, completely known          *)
          k         (* the continuation to build the code. Must be applied  *)
                    (* to a substitution                                    *)
          msigma    (* the substitution that is built when matching occurs  *)
          term      (* code for the term being matched against the pattern  *)
          =
case (msigma) of
   NONE => k NONE
 | SOME (sigma) =>
(case pat of
    Var u =>
      (case find u sigma of
         NONE =>
           k (SOME ((u,term) :: sigma))
       | SOME w =>
            <if termeq ~w ~term
                then ~(k (SOME sigma))
                else ~(k NONE)>)
  | Term(Int n) =>
      <case ~(unWrapCode term) of
         Int u => if u= ~(lift n)
                     then ~(k msigma)
                     else ~(k NONE)
       | _ => ~(k NONE)>
```

```
  | Term(Op(p1,s1,p2)) =>
      <case ~(unWrapCode term) of
         Op(t1,s2,t2) =>
            if ~(lift s1) = s2
               then ~(match p1
                           (fn msig => match p2 k msig <t2>)
                           msigma
                           <t1>)
               else ~(k NONE)
        | _ => ~(k NONE)>
)
```

Rewriting builds the code of a term from a rule. We do this as follows:

```
fun rewrite (lhs,rhs) =
<fn (Wrap t) =>
   ~(match lhs
           (fn NONE => <Wrap t>         (* the initial continuation *)
             | SOME s => subst s rhs)
           (SOME [])
           <Wrap t>)>;
```

Note how we build a continuation to apply the substitution to the right-hand side of the rule, and pass it to the `match` function. If the continuation is ever passed the failure substitution (`NONE`) it simply returns the original term.

When we apply the `rewrite` function to a rule some code is constructed.

```
val r3 = (** (x + y) + z   =>  x + (y + z)  **)
  (Term(Op(Term (Op(Var "x","+",Var "y")), "+", Var "z")),
   Term(Op(Var "x","+",Term(Op(Var "y","+",Var "z")))));

-| rewrite r3;
val it =
<(fn Wrap a =>
     (case a of
       Op(d,c,b) =>
          if "+" %= c
             then (case %unWrap d of
                    Op(g,f,e) =>
                       if "+" %= f
                          then Wrap (Op(g,"+",Wrap (Op(e,"+",b))))
                          else Wrap a
                  | _  =>
                       Wrap a)
             else Wrap a
      | _  => Wrap a))>
 : <term  -> term >
```

6 Safe reductions under brackets

The purpose of MetaML is to control evaluation order. The bracket annotation is the mechanism used to delay evaluation. It is used to say "*do not apply any reduction rules in this term until I say so*". Even so, there *are* reduction rules that are safe to apply even inside brackets. These rules never change the semantics or the termination properties of term, or the order in which sub-terms are evaluated. The reason we wish to apply such rules is that they can significantly reduce the size and complexity of a piece of code without affecting any of its important properties. To write multi-stage programs effectively, one needs to observe the programs produced, and these programs should be as simple as possible. For this reason, it is important that code produced be as simple as possible.

6.1 Safe-beta

There is one safe case which is particularly well known, namely instances of Plotkin's β_v rule [9]. Whenever an application is constructed where the function part is an explicit lambda abstraction, and the argument part is a value, then that application can be symbolicly beta reduced. In order to avoid duplicating code we restrict our safe-beta reductions to those terms where the argument is a constant or a variable (while Plotkin's β_v rule also allows the values to be lambda expressions). For example in:

```
val g = <fn x => x * 5>;
val h = <fn x => (~g x) - 2>;
```

The variable h evaluates to: `<fn d1 => (d1 * 5) - 2>` rather than
`<fn d1 => ((fn d2 => d2 * 5) d1) - 2>`.

6.2 Safe-eta

Another simple example is eta-reduction, i.e terms of the form: `(fn x => e x)` where `e` is a value (an explicit lambda or a variable) and `x` does not occur free in `e`. Such terms can be eta-reduced to `e` without changing their meaning or termination behavior. To see how this works in MetaML see the example below:

```
-| <fn f => (fn x => f x)>;
val it = <(fn a => a)> : ['b,'c].<('c  -> 'b ) -> 'c  -> 'b>

-| <fn (f,y) => (fn x => (f y) x)>;
val it = <(fn (b,a) => (fn c => b a c))> :
         ['b,'c,'d].<(('d  -> 'c  -> 'b ) * 'd ) -> 'c  -> 'b>
```

Notice how the eta-rule is applied in the first example, but not in the second. This is because the conditions for safety are not met (the function part is an application not a value) in the second example.

6.3 Safe-let-hoisting

Let-hoisting is illustrated by the following examples:

```
-| <let val x = (let val y = 5 in y+1 end) in x + 2 end>;
val it = <let val a = 5 val b = a %+ 1  in b %+ 2 end> : <int>

-| <let val y = 5 in let val x = y+1 in x+2 end end>;
val it = <let val a = 5 val b = a %+ 1  in b %+ 2 end> : <int>
```

Safe-beta, safe-eta, and safe-let-hoisting are instances of Wadler and Sabry's call-by-value equivalence rules [10]. Applying these rules makes it harder to understand *why* a particular program was generated, but in our experience, the resulting programs are smaller, simpler, and easier to understand. These advantages make this tradeoff worthwhile.

7 Non-standard Extensions

We built MetaML to investigate new paradigms of programming. As we used MetaML we discovered several simple extensions to ML in addition to the staging annotations that were quite useful. These extensions are not not original to MetaML. All are well thought-out ideas that have appeared in the literature, and several have appeared as features in other languages.

7.1 Higher order type constructors

It is sometimes useful to define a parameterized type constructor, parameterized not just by a type, but by another type constructor. In MetaML this can be done by placing a kind annotation on the parameter that indicates it is a type constructor. For example consider the definition below for a tree of integers with an arbitrary branching factor:

```
datatype ('F : * -> * ) tree = tip of int | node of ('F tree) 'F;

datatype 'a binary = bin of 'a * 'a;
```

The branching factor of a `tree` is specified by the parameter `'F` which is itself a type constructor. The notation: `'F : * -> *` means `'F` has kind "type to type", which means it is a type constructor taking one type (`*`) to another type (`*`).

For example the tree: `node(bin(tip 4, tip 7))` has type `binary tree`, and the tree constructed by: `node[tip 4, tip 0, tip 6]` has type `list tree`.

It is possible to define type constructors parameterized by several higher order type constructors by definitions of the form:
`datatype ('F : * -> *, 'G: * -> *) T = ...`
or by a type constructor that takes several arguments by a definition of the form:
`datatype ('F : * -> * -> * ) S = ....`

The postfix application (e.g. `(int,string) T`) of type constructors in ML causes a little subtlety. A unary type constructor can be constructed from a binary type constructor by partial application. But this requires some special syntax since in ML all arguments are "grouped together" inside parentheses. We think of the normal (parenthesized) notation of ML as a shorthand for our special syntax which allows partial application. Thus the normal `('a,'b,'c) T` is a shorthand for the more verbose (but more flexible since it allows partial application) `'a ('b ('c T))`.

For example by defining the binary type constructor `state`

```
datatype ('a,'b) state = St of ('a -> ('b * 'a));
```

we can construct an instance: `node(St(fn x => (4,x)))`, which has type `(int state) tree` where the parameter to `tree` is a partial application of `state`

7.2 Local polymorphism

It is often convenient to build records where a component of a record is a polymorphic function. This allows a limited form of "local polymorphism". By "local" we mean non-Hindley-Milner because all of the `forall` quantifications are not at the outermost level.

For example consider the specification of a list monoid as a record containing three polymorphic elements: an injection function, a plus function, and a zero element. In MetaML we specify this with an extension to the datatype definition which allows polymorphic record components:

```
datatype list_monoid = LM of
 { inject : ['a].'a -> 'a list,
   plus : ['a]. 'a list -> 'a list -> 'a list,
   zero : ['a].'a list
 };
```

The notation `inject: ['a]. 'a -> 'a list` declares that the inject component of the record must be a polymorphic function.

We could construct an instance of list_monoid by:

```
val lm1 = LM{inject = fn x => [x],
             plus   = fn x => fn y => x@y,
             zero   = []}
```

We can exploit the polymorphism of the record by using pattern matching:

```
fun f (LM{inject=inj, plus = sum, zero = z}) =
      (sum z (inj 2), sum (inj true) (inj false));
```

When applied to `list_monoid` we obtain:

```
-| f lm1;
val it = ([2],[true ,false ]) : (int list  * bool  list )
```

Note that the `sum` and `inj` functions are used polymorphically. Because of the explicit type annotations in the `datatype` declaration MetaML knows to generalize polymorphic names introduced by pattern matching and to enforce that construction of such records is only allowed on truly polymorphic objects. The effect of local polymorphism and higher order type constructors on the Hindley-Milner type inference system has been well studied[3, 7].

7.3 Monads

A monad is a type constructor M (a type constructor is a function on types, which given a type produces a new type), and two polymorphic functions *unit : 'a* $\rightarrow$ *('a M)* and *bind: ('a M)* $\rightarrow$ *('a* $\rightarrow$ *'b M)* $\rightarrow$ *('b M)*. The usual way to interpret an expression with type *'a M* is as a computation which represents a potential action that also returns a value of type *'a*.

Actions might include things like performing I/O, updating a mutable variable, or raising an exception. It is possible to emulate such actions in a purely functional setting by explicitly threading "stores", "I/O streams", or "exception continuations" in and out of all computations. We sometimes call such an emulation the *reference implementation*, since it describes the actions in a purely functional manner, though it may be inefficient.

The two polymorphic functions *unit* and *bind* must meet the following three axioms:

$$
\begin{array}{ll}
\text{(left id)} & \mathit{bind}\ (\mathit{unit}\ x)\ (\lambda y.e[y]) = e[x/y] \\
\text{(right id)} & \mathit{bind}\ e\ (\lambda y.\mathit{unit}\ y) = e \\
\text{(bind assoc)} & \mathit{bind}\ (\mathit{bind}\ e\ (\lambda x.f[x]))\ (\lambda y.g[y]) = \mathit{bind}\ e\ (\lambda z.\mathit{bind}\ (f[z/x])(\lambda w.g[w/y]))
\end{array}
$$

where on the left side of an equation $e[x]$ indicates that e is an expression that contains occurrences of the free variable x, and on the right side of an equation $e[x/y]$ means substitute y for all free occurrences of x in e.

The monadic operators, *unit* and *bind*, are called the standard morphisms of the monad, and are used to create empty actions (*unit*), and sequence two actions (*bind*). A particular monad must also have non-standard morphisms that describe the primitive actions of the monad (like *fetch* the value from a variable and *update* a variable in the monad of mutable state).

A useful property of monads is that they encapsulate their actions in an abstract datatype (ADT), where the only access to the encapsulation is through *unit*, *bind*, and the non-standard morphisms. Like any ADT, it is possible to use different implementations without affecting the behavior of the system built on top of the ADT. Thus it is possible for a purely functional language to use a primitive implementation of a monad that actually side-effects the world[5], and for the applications built on top of this ADT to still appear purely functional. As long as the primitive implementation behaves like the reference implementation (that might passes stores etc.) everything works out.

Monads perform two useful functions. First, they abstract away all the "plumbing" that all the explicit threading implies, and second, they make explicit actions that can be used to effect the world.

7.4 Monads in MetaML

In MetaML a monad is a data structure encapsulating the type constructor M and the *unit* and *bind* functions[14].

```
datatype ('M : * -> * ) Monad = Mon of
    ( ['a]. 'a -> 'a 'M) * (['a,'b]. 'a 'M -> ('a -> 'b 'M) -> 'b 'M);
```

This definition uses two of the other non-standard extensions to ML. First, it declares that the argument `('M : * -> *)` of the type constructor `Monad` is itself a unary type constructor. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions.

In MetaML, `Monad` is a first-class, although *pre-defined* type. In particular, there are two syntactic forms which are aware of the `Monad` datatype: `Do` and `Return` [5]. Both are parameterized by an expression of type `'M Monad`. Users may freely construct their own monads, though they must be careful that their instantiation meets the monad axioms listed above.

`Do` is MetaML's interface to the monadic *bind* and `Return` is MetaML's interface to the monadic *unit*. In MetaML these are really nothing more than syntactic sugar for the following:

Syntactic Sugar		*Derived Form*
`Do (Mon(unit,bind)) { x <- e; f }`	=	`bind e (fn x => f)`
`Return (Mon(unit,bind)) e`	=	`unit e`

In addition the syntactic sugar of the `Do` allows a sequence of $\mathtt{x}_i$ `<-` $\mathtt{e}_i$ forms, and defines this as a nested sequence of `Do`'s. For example:

```
Do m { x1 <- e1; x2 <- e2 ; x3 <- e3 ; e4 }   =
   Do m { x1 <- e1; Do m { x2 <- e2 ; Do m { x3 <- e3 ; e4 }}}
```

7.5 An example monad

A simple example is the `intState` monad which encapsulates read and write actions on a single, mutable, integer variable. We give a reference implementation which encodes the mutable integer value as an integer. This integer is threaded into and out of every computation. `Read`'s will access the value, and `Write`'s will pass out a new updated value.

```
datatype 'a St = St of (int -> ('a * int));
fun unSt (St f) = f;

(* unit : 'a -> St 'a *)
```

```
fun unit x = St(fn n => (x,n));

(* bind : (St 'a) -> ('a -> St 'b) -> (St 'b) *)
fun bind e f = St(fn n => let val (a,n') = (unSt e) n
                          in unSt(f a) n' end);

val intState : St Monad = Mon(unit,bind);
```

We encapsulate the type constructor of the monad as the algebraic datatype `St`, the regular morphisms as functions over this datatype, and then encapsulate them with the `Mon` data constructor.

The non-standard morphisms of the `intState` monad are the actions `read` and `write`. Because there is only *one* variable they need not take a variable as an argument.

```
(* read : int St *)
val read = St(fn n => (n,n));

(* write : int -> unit St *)
fun write n' = St(fn n => ( (), n' ));
```

It is interesting to unfold all these definitions by hand on a simple example:

```
Do intState { x <- read ; write (x+1) } =

Do (Mon(unit,bind)) { x <- read ; write (x+1) }  =

bind read (fn x => write (x+1))  =

St(fn n => let val (a,n') = (unSt read) n  in unSt(write (a+1)) n' end)  =

St(fn n => let val (a,n') = (n,n)  in ( (), n'+1 ) end)
```

There are three important things to notice about this example. First, by writing it in monadic style, the sequencing (the `read` *before* the `write`) is enforced by the data dependencies of the result. Second, the "plumbing" of passing the `int` valued state is completely abstracted away in the source. Third, it makes the `read` and `write` "actions" that must be performed explicit.

7.6 Safe Monad-law-normalization inside brackets

Like safe-beta, safe-eta, and safe-let-hoisting the monad laws are reduction rules that can safely be applied inside brackets without changing the evaluation order, termination behaviour, or any other semantic property. We give several examples below which illustrate the effect of monad law normalization on constructed code:

```
(* left id *)
-| <Do intState { x <- Return intState 5; Return intState x + 2 }>;
```

```
val it = <Return %intState 5 %+ 2> : <int St>

(* right id *)
-| <fn e => Do intState { x <- e; Return intState x}>;
val it = <(fn a => a)> : ['b].<'b  St  -> 'b  St>
```

When monadic code is constructed, the monad normalization laws are automatically applied in the MetaML interpreter.

8 From Interpetors to compilers using staging

In this section, we construct a compiler by annotating a monadic interpretor for a small imperative *while-language*. We proceed in two steps. First, we introduce the language and its denotational semantics by giving a monadic interpreter as a one stage MetaML program. Second, we stage this interpreter by using a two stage MetaML program in order to produce a compiler.

8.1 The while-language

In this section, we introduce a simple *while-language* composed from the syntactic elements: expressions (`Exp`) and commands (`Com`). In this simple language expressions are composed of integer constants, variables, and operators. A simple algebraic datatype to describe the abstract syntax of expressions is given in MetaML below:

```
datatype Exp =
  Constant of int              (*   5     *)
| Variable of string           (*   x     *)
| Minus of (Exp * Exp)         (*  x - 5  *)
| Greater of (Exp * Exp)       (*  x > 1  *)
| Times of (Exp * Exp) ;       (*  x * 4  *)
```

Commands include assignment, sequencing of commands, a conditional (*if* command), while loops, a print command, and a declaration which introduces new statically scoped variables. A declaration introduces a variable, provides an expression that defines its initial value, and limits its scope to the enclosing command. A simple algebraic datatype to describe the abstract syntax of commands is:

```
datatype Com =
  Assign of (string * Exp)         (*  x := 1                      *)
| Seq of (Com * Com)               (*  { x := 1; y := 2 }          *)
| Cond of (Exp * Com * Com)        (*  if x then x := 1 else y := 1 *)
| While of (Exp * Com)             (*  while x>0 do x := x - 1      *)
| Declare of (string * Exp * Com)  (*  declare x = 1 in x := x - 1  *)
| Print of Exp;                    (*  print x                      *)
```

A simple while-program in concrete syntax, such as

```
declare x = 150 in
   declare y = 200 in { while x > 0 do { x := x - 1; y := y - 1 }; print y}
```

is encoded abstractly in these datatypes as follows:

```
val S1 =
Declare("x",Constant 150,
  Declare("y",Constant 200,
    Seq(While(Greater(Variable "x",Constant 0),
              Seq(Assign("x",Minus(Variable "x",Constant 1)),
                  Assign("y",Minus(Variable "y",Constant 1)))),
        Print(Variable "y"))));
```

8.2 The structure of the solution

Staging is an important technique for developing efficient programs, but it requires some forethought. To get the best results one should design algorithms with their staged solutions in mind.

The meaning of a while-program depends only on the meaning of its component expressions and commands. In the case of expressions, this meaning is a function from environments to integers. The environment is a mapping between names (which are introduced by `Declare`) and their values.

There are several ways that this mapping might be implemented. Since we intend to stage the interpreter, we break this mapping into two components. The first component, a list of names, will be completely known at compile-time. The second component, a list of integer values that behaves like a stack, will only be known at the run-time of the compiled program.

The functions that access this environment distribute their computation into two stages. First, determining at what location a name appears in the name list, and second, by accessing the correct integer from the stack at this location. In a more complicated compiler the mapping from names to locations would depend on more than just the declaration nesting depth, but the principle remains the same. Since every variable's location can be completely computed at compile-time, it is important that we do so, and that these locations appear as constants in the next stage.

Splitting the environment into two components is a standard technique (often called a binding time improvement) used by the partial evaluation community[4]. We capture this precisely by the following purely functional implementation.

```
type location = int;
type index = string list;
type stack = int list;

(* position : string -> index -> location *)
fun position name index =
    let fun pos n (nm::nms) = if name = nm then n else pos (n+1) nms
    in pos 1 index end;
```

```
(* fetch : location -> stack -> int *)
fun fetch n (v::vs) = if n = 1 then v else fetch (n-1) vs;

(* put: location -> int -> stack -> stack *)
fun put n x (v::vs) = if n = 1 then x::vs else v::(put (n-1) x vs);
```

The meaning of `Com` is a stack transformer and an output accumulator. It transforms one stack (holding the values of the variables in scope) into another stack (with presumably different values for the same variables) while accumulating the output printed by the program.

To produce a monadic interpreter we could define a monad which encapsulates the index, the stack, and the output accumulation. Because we intend to stage the interpreter we do not encapsulate the index in the monad. We want the monad to encapsulate only the dynamic part of the environment (the stack of values where each value is accessed by its position in the stack, and the output accumulation).

The monad we use is a combination of *monad of state* and the *monad of output*.

The part corresponding to the monad of state is similar to the monad described in section 7.5, except the mutable value is not an integer, but instead a vector of mutable integers that will be managed like a stack.

```
datatype 'a M = StOut of (int list -> ('a * int list * string));
fun unStOut (StOut f) = f;
fun unit x = StOut(fn n => (x,n,""));
fun bind e f = StOut(fn n => let val (a,n1,s1) = (unStOut e) n
                                 val (b,n2,s2) =  unStOut(f a) n1
                             in (b,n2,s1 ^ s2) end);
val mswo : M Monad = Mon(unit,bind); (* Monad of state with output *)
```

The non-standard morphisms must describe how the stack is extended (or shrunk) when new variables come into (or out of) scope; how the value of a particular variable is read or updated; and how the printed text is accumulated. Each can be thought of as an action on the stack of mutable variables, or an action on the print stream.

```
(* read : location -> int M *)
fun read i = StOut(fn ns => (fetch i ns,ns,""));

(* write : location -> int -> unit  M *)
fun write i v = StOut(fn ns =>( (), put i v ns, "" ));

(* push: int -> unit  M *)
fun push x = StOut(fn ns => ( (), x :: ns, ""));

(* pop : unit M *)
val pop = StOut(fn (n::ns) => ((), ns, ""));

(* output: int -> unit M *)
fun output n = StOut(fn ns => ( (), ns, (toString n)^" "));
```

8.3 Step 1: monadic interpreter

Because expressions do not alter the stack, or produce any output, we could give an evaluation function for expressions which is not monadic, or which uses a simpler monad than the monad defined above. We choose to use the monad of state with output throughout our implementation for two reasons. One, for simplicity of presentation, and two because if the while language semantics should evolve, using the same monad everywhere makes it easy to reuse the monadic evaluation function with few changes.

The only non-standard morphism evident in the `eval1` function is `read`, which describes how the value of a variable is obtained. The monadic interpretor for expressions takes an index mapping names to locations and returns a computation producing an integer.

```
(*  eval1: Exp -> index -> int M *)
fun eval1 exp index =
case exp of
  Constant n => Return mswo n
| Variable x => let val loc = position x index
                in read loc end
| Minus(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (a - b) }
| Greater(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (if a '>' b then 1 else 0) }
| Times(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (a * b) };
```

The interpreter for `Com` uses the non-standard morphisms `write`, `push`, and `pop` to transform the stack and the morphism `output` to add to the output stream.

```
(* interpret1 : Com -> index -> unit M *)
fun interpret1 stmt index =
case stmt of
  Assign(name,e) =>
   let val loc = position name index
   in Do mswo { v <- eval1 e index ;  write loc v } end
| Seq(s1,s2) =>
   Do mswo { x <- interpret1 s1 index;
             y <- interpret1 s2 index;
             Return mswo () }
| Cond(e,s1,s2) =>
   Do mswo { x <- eval1 e index;
```

```
            if x=1
               then interpret1 s1 index
               else interpret1 s2 index }
| While(e,body) =>
   let fun loop () =
        Do mswo { v <- eval1 e index ;
                  if v=0 then Return mswo ()
                         else Do mswo { interpret1 body index ;
                                        loop () } }
   in loop () end
| Declare(nm,e,stmt) =>
   Do mswo { v <- eval1 e index ;
            push v ;
            interpret1 stmt (nm::index);
            pop }
| Print e =>
   Do mswo { v <- eval1 e index;
            output v };
```

Although `interpret1` is fairly standard, we feel that two things are worth pointing out. First, the clause for the `Declare` constructor, which calls `push` and `pop`, implicitly changes the size of the stack and explicitly changes the size of the index (`nm:index`), keeping the two in synch. It evaluates the initial value for a new variable, extends the index with the variables name, and the stack with its value, and then executes the body of the `Declare`. Afterwards it removes the binding from the stack (using `pop`), all the while implicitly threading the accumulated output. The mapping is in scope only for the body of the declaration.

Second, the clause for the `While` constructor introduces a local tail recursive function `loop`. This function emulates the body of the while. It is tempting to control the recursion introduced by the `While` by using the recursion of the `interpret1` function itself by using a clause something like:

```
| While(e,body) =>
   Do mswo { v <- eval1 e index ;
            if v=0 then Return mswo ()
                   else Do mswo { interpret1 body index ;
                                  interpret1 (While(e,body)) index }
          }
```

Here, if the test of the loop is true, we run the body once (to transform the stack and accumulate output) and then repeat the whole loop again. This strategy, while correct, will have disastrous results when we stage the interpreter, as it will cause the first stage to loop infinitely.

There are two recursions going on here. First the unfolding of the finite data structure which encodes the program being compiled, and second, the recursion in the program being compiled. In an unstaged interpreter a single loop suffices. In a staged interpreter, both loops are necessary. In the first stage we only unfold the program being compiled and this must always terminate. Thus we must plan ahead as we follow our three step process. Nevertheless, despite the concessions we have made to staging, this interpreter is still clear, concise and describes the semantics of the while-language in a straight-forward manner.

8.4 Step 2: staged interpreter

To specialize the monadic interpreter to a given program we add two levels of staging annotations. The result of the first stage is the intermediate code, that if executed returns the value of the program. The use of the bracket annotation enables us to describe precisely the code that must be generated to run in the next stage. Escape annotations allow us to escape the recursive calls of the interpreter that are made when compiling a while-program.

```
(* eval2: Exp -> index -> <int M> *)
fun eval2 exp index =
case exp of
  Constant n => <Return mswo ~(lift n)>
| Variable x =>
   let val loc = position x index
   in <read ~(lift loc)> end
| Minus(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (a - b) }>
| Greater(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (if a '>' b then 1 else 0) }>
| Times(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (a * b) }>;
```

The `lift` operator inserts the value of `loc` as the argument to the `read` action. The value of `loc` is known in the first-stage (compile-time), so it is transformed into a constant in the second-stage (run-time) by `lift`.

To understand why the escape operators are necessary, let us consider a simple example: `eval2 (Minus(Constant 3,Constant 1)) []`. We will unfold this example by hand below:

```
eval2 (Minus(Constant 3,Constant 1)) [] =

< Do mswo
     { a <- ~(eval2 (Constant 3) []);
       b <- ~(eval2 (Constant 1) []);
       Return mswo (a-b)} >                   =
```

```
< Do mswo
     { a <- ~<Return mswo 3>;
       b <- ~<Return mswo 1>;
       Return mswo (a - b)} >                  =

< Do mswo
     { a <- Return mswo 3;
       b <- Return mswo 1;
       Return mswo (a - b)} >                  =

< Do %mswo
     { a <- Return %mswo 3;
       b <- Return %mswo 1;
       Return %mswo (a %- b)} >
```

Each recursive call produces a bracketed piece of code which is spliced into the larger piece being constructed. Recall that escapes may only appear at level-1 and higher. Splicing is axiomatized by the the first elimination rule for code: `~<x>` $\longrightarrow$ `x`, which applies only at level-1. The final step, where `mswo` and `-` become `%mswo` and `%-`, occurs because both are free variables and are lexically captured.

Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

```
(*  interpret2 : Com -> index ->  <unit M> *)
fun interpret2 stmt index =
case stmt of
  Assign(name,e) =>
   let val loc = position name index
   in <Do mswo { n <- ~(eval2 e index) ;
                 write ~(lift loc) n }>
   end
| Seq(s1,s2) =>
   <Do mswo { x <- ~(interpret2 s1 index);
              y <- ~(interpret2 s2 index);
              Return mswo () }>
| Cond(e,s1,s2) =>
   <Do mswo { x <- ~(eval2 e index);
              if x=1
                 then ~(interpret2 s1 index)
                 else ~(interpret2 s2 index)}>
| While(e,body) =>
   <let fun loop () =
            Do mswo { v <- ~(eval2 e index);
                    if v=0
                       then Return mswo ()
                       else Do mswo { q <- ~(interpret2 body index); loop ()}
```

```
              }
    in loop () end>
| Declare(nm,e,stmt) =>
   <Do mswo { x <- ~(eval2 e index) ;
              push x ;
              ~(interpret2 stmt (nm::index)) ;
              pop }>
| Print e =>
   <Do mswo { x <- ~(eval2 e index) ;
              output x }>;
```

An example. The function `interpret2` generates a piece of code from a `Com` object. To illustrate this we apply it to the simple program: `declare x = 10 in { x := x - 1; print x }` and obtain:

```
<Do %mswo
    { a <- Return %mswo 10
    ; %push a
    ; Do %mswo
         { e <- Do %mswo
                   { d <- Do %mswo
                             { b <- %read 1
                             ; c <- Return %mswo 1
                             ; Return %mswo b %- c
                             }
                   ; %write 1 d
                   }
         ; g <- Do %mswo
                   { f <- %read 1
                   ; %output f
                   }
         ; Return %mswo ()
         }
    ; %pop
   }>
```

By applying the safe monad normalization laws while constructing the above program we obtain the more satisfying:

```
<Do %mswo
    { %push 10
    ; a <- %read 1
    ; b <- Return %mswo a %- 1
    ; c <- %write 1 b
    ; d <- %read 1
    ; e <- %output d
    ; Return %mswo ()
    ; %pop
    }>
```

The difference in the complexity of the two programs illustrates why program normalization is important if constructed programs are to be observed. In the MetaML implementation the normalization laws can be turned on and off. They are all on by default. The side effecting function `feature` can be used to control the normalization laws. `feature 0` displays the normalization modes, and `feature n` toggles the `nth` feature.

```
-| feature 0;
1 Safe-beta is on.
2 Safe-eta is on.
3 Let-hoisting is on.
4 Monad-law-normalization is on.
val it = false  : bool

-| feature 4;
Monad-law-normalization is off.
val it = false  : bool
```

9 Typing staged programs

In Figure 9 the derivation rules for typing a subset of MetaML are given. The interesting rules are **Br n** which addresses the typing of bracketed expressions, **Esc n+1** which addresses the typing of escaped expressions, and **Run n**. Note that **Br n** raises the level n of the term bracketed, and that **Esc n+1** only applies at levels 1 and higher. This ensures that escaped expressions only appear inside brackets.

9.1 Type questions still to be addressed

The type system presented in Figure 9 represents the type system in the MetaML implementation. This type system has some drawbacks. In particular it types the program `<fn x => ~ (run <x>)>` which leads to a runtime error.

```
-| <fn x => ~(run <x>)>;
Error: The term:
x
in file 'top level' 18 - 19
variable bound in phase 1 used too early in phase 1
```

This is because the rule **Run n** removes brackets without lowering the level n. This is the normal course of affairs, and is the right thing to do, except if `run` is applied to a piece of code with free variables which are bound at a level higher than the level at which `run` is executed. This is the case for the example above because `x` is bound at level 1, and `run` is executed at level 0.

Designing a type system to keep track of this is quite hard. We have designed several type systems to invalidate such programs[11, 13]. Unfortunately

Domains and Relations

levels	$n \rightarrow 0 \mid 1 \mid n+1 \mid n+2 \mid \ldots$
integers	$i \rightarrow \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$
types	$\tau \rightarrow \mathtt{int} \mid \tau \rightarrow \tau \mid \langle \tau \rangle \mid \alpha$
terms	$e \rightarrow i \mid x \mid e\, e \mid \lambda x^{\tau} . e \mid \mathtt{<}e\mathtt{>} \mid \tilde{\ } e \mid \mathtt{run}\, e \mid \uparrow v$
type environments	$\Delta \rightarrow \circ \mid \Delta, x \mapsto (\tau, \alpha)^n$

where $(\Delta, x \mapsto (\tau)^n)y \equiv$ if $x = y$ then $(\tau, \alpha)^n$ else $\Delta\ y$

term typing *at level n* $\Delta \overset{n}{\vdash} e : \tau$

Static Semantics

Int n: $\Delta \overset{n}{\vdash} i : \mathtt{int}$

Var: $$\frac{\Delta\, x = (\tau)^i \quad i \leq n}{\Delta \overset{n}{\vdash} x : \tau}$$

Br n: $$\frac{\Delta \overset{n+1}{\vdash} e : \tau}{\Delta \overset{n}{\vdash} \mathtt{<}e\mathtt{>} : \langle \tau \rangle}$$

Esc n+1: $$\frac{\Delta \overset{n}{\vdash} e : \langle \tau \rangle}{\Delta \overset{n+1}{\vdash} \tilde{\ } e : \tau}$$

App n: $$\frac{\Delta \overset{n}{\vdash} e_1 : \tau_1 \rightarrow \tau \qquad \Delta \overset{n}{\vdash} e_2 : \tau_1}{\Delta \overset{n}{\vdash} e_1\ e_2 : \tau}$$

Abs n: $$\frac{\Delta, x \mapsto (\tau_1)^n \overset{n}{\vdash} e : \tau_2}{\Delta \overset{n}{\vdash} \lambda x^{\tau_1} . e : \tau_1 \rightarrow \tau_2}$$

Run n: $$\frac{\Delta \overset{n}{\vdash} e : \langle \tau \rangle}{\Delta \overset{n}{\vdash} \mathtt{run}\, e : \tau}$$

Fig. 1. The Static Semantics of MetaML

these systems either also throw away other good programs, or require elaborate annotations.

In the MetaML implementation we take the position that such errors are similar to errors such as taking the head or tail of an empty list: Typeable, but leading to a runtime error. Avoiding such errors are the responsibility of the programmer. We have written many staged programs and always avoided this error.

Research into improving the type system is an area of continued research.

10 Conclusion

A staged programming language gives the programmer a new paragdigm for constructing efficient programs. We have illustrated this by building staged programs for interpreters and polytypic programs.

We have also found that several other"advanced" features such as higher-order type constructors, local polymorphism, and monads have many uses.

We believe that languages with these features help programmers construct programs which are easier to maintain because they are generic, yet they are still efficient.

11 Exercises

- **Staged member function.** Write a staged membership function, where the list is known in the first stage, but the element being searched for is not known till the second phase. Experiment with the use of the `lift` annotation to make your generated code more readable.
- **3 level inner product function.** The inner product function can be staged in three stages. The 1 stage innner product function is given below.

```
fun inner_prod n x y =
if n = 0
   then 1
   else (nth n x)*(nth n y) + inner_prod (n-1) x y;
```

  The three stage function is written to proceed as follows: In the first stage the arrival of the size of the vectors offers an opportunity to specialize the inner product function on that size, removing the overhead of looping over the body of the computation n times. The arrival of the first vector affords a second opportunity for specialization. If the inner product of that vector is to be taken many times with other vectors it can be specialized by removing the overhead of looking up the elements of the first vector each time. This is exactly the case when computing the multiplication of 2 matrixes. For each row in the first matrix, the dot product of that row will be taken for each column of the second. In addition the second stage affords the opportunity for additional optimization. Since the first vector is known multiplications by 1 or 0 can be elminated in the third stage.

– **Simple Compiler.** Define a language. Write an interpreter for the language. Stage the interpreter to construct a compiler.
– **Post code generation optimization.** Use the pattern matching for code feature of MetaML to construct a simple code optimization phase.

References

1. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *20thACM Symposium on Principles of Programming Languages*, pages 493–501, January 1993.
2. Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 65:122–157, 1985.
3. Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polumorphism. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, June 1993. ACM Press.
4. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Series editor C. A. R. Hoare. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
5. John Launchbury and Simon Peyton-Jones. Lazy functional state threads. In *PLDI'94: Programming Language Design and Implementation, Orlando, Florida*, pages 24–35, New York, June 1994. ACM Press.
6. Matthieu Martel and Tim Sheard. Introduction to multi-stage programming using metaml. Technical report, OGI, Portland, OR, September 1997.
7. Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 54–67, January 1996.
8. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
9. G. D. Plotkin. Call-by-name, call-by-value- and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
10. Amr Sabry and Philip Wadler. A reflection on a call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, November 1997.
11. Walid Taha, Zine-el-abidine Benaissa, and Tim Sheard. The essence of staged programming. Technical report, OGI, Portland, OR, December 1997.
12. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.
13. Walid Taha and Tim Sheard. Metaml: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, To Appear.
14. Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.