Introduction to multistage Programming

Using MetaML

Tim Sheard Zino Benaissa Matthieu Martel

Revision 2.

Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
P.O. Box 91000
Portland, OR 97291-1000

February 2000

The research reported in this paper was supported by the USAF Air Material Command, contract # F19628-96-C-0161, NSF Grants CCR-9803880 and IRI-9625462, and the Department of Defense.

Contents

Introduction to Multi-Stage Programming Using MetaML

Tim Sheard and Matthieu Martel
Revision 2.

Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
P.O. Box 91000
Portland, OR 97291-1000

February 2000

1 Introduction

In a meta-programming system meta-programs manipulate object-programs. Meta-programs may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs, and execute object-programs to obtain their values.

There are two important kind of meta-programming scenarios: program generators, and program analyses . MetaML is a meta-programming system designed for the first scenario: program generation .

A program generator (a meta-program) solves a particular problem by constructing another program (an object-program) that solves the problem at hand. Usually the generated (object) program is "specialized" for the particular problem and uses less resources than a general purpose, non-generator solution. This method of producing specialized solutions by using a program generator is often called *staged programming*. A meta-programming system is one tool that can be used when practicing the staged programming method (a partial evaluator is another tool).

MetaML is intended to be a conservative extension[1] of Standard SML which allows the programmer to write multistage programs. The basic annotations used to describe a staged code are *meta-brackets escape* . Bracketing an expression indicates that its computation is to be delayed to the next stage, i.e. part of the object program being generated. Evaluation may be delayed for many stages by bracketing the corresponding expression as many times as wished. One may also decrease the execution stage of an expression by *escaping* it from the freeze imposed by the

[1]At this time this is not quite true, our implementation is not a complete implementation of Standard ML . There are also several small syntactic differences, see section 3.2.

brackets. This role is played by the escape operator (~). Other more sophisticated operators are described later in this document.

Writing correct multistage programs quickly becomes a challenging task (a taste of that is offered to the reader in Section 6). MetaML helps the user by type checking both the level and the traditional type of expressions. Type-checking is performed once and for all before any execution occurs. Type-checking succeeds only if every expression at every stage is well-typed. So, if the type-checker does not complain, the program won't go wrong at any stage of its execution. In addition, the system checks the soundness of *binding-times*. Broadly speaking, it prevents an expression e computed at level l from being used in another expression at level less than l (i.e. before e is computed).

This document attempts to be an introduction to both multistage programming and MetaML. Section 2 describes the features of both SML (for those not familiar with SML) and its meta-level extension. In Section 6, we examine, step by step, the design of two larger multistage programs: a *merge* function over lists and a *shortest-path* algorithm.

2 Language overview

MetaML is a meta-programming system . In MetaML one writes meta-programs in a meta-language which manipulate object-programs in an object-language . In MetaML both the meta-language and object-languages are SML [MTHM97, Pau96].

Thus every ordinary SML program is also a MetaML program; such programs just happen to do no object-level manipulation. MetaML is intended to be a conservative extension to SML. It has the same syntax and semantics with the addition of a few meta-level programming constructs.

2.1 Basic SML constructs

For the SML novice we give an extremely brief overview of the ordinary features of SML. In the next section we explain the additional meta-level features. Those familiar with the syntax and semantics of Standard SML may want to skip directly to Section 2.2.

2.1.1 MetaML's top-level interactive loop

SML and MetaML's interface to the user is an *interactive loop*. In this interactive loop, the user supplies some input. It is parsed, type-checked, and if the input is type-checked successfully, the input is evaluated, and the result is printed for the user to inspect. In our implementation starting the system provides the following:

```
Welcome to MetaML...

-|
```

The prompt "`-|`" indicates to the user that the system is waiting for input. The user submits expressions to be evaluated by typing them into the system and terminating them by a semicolon ";" followed by a *newline*. MetaML reads, type-checks, and evaluates the current expression. The result is then displayed and a new prompt appears. MetaML accepts different kinds of expressions, which are described by the syntax of SML [MTHM97] plus MetaML's meta-extensions.

For example, constants are the simplest expressions in the language.

```
-| 2;
val it = 2 : int

-| 2+3;
val it = 5 : int
```

Above, in the first example the input `2;` is parsed, type-checked to have type `int`, and then evaluated. In the second example the expression `2+3` is treated similarly, but here we see that the expression is evaluated to `5` before its value is printed;

Note that expressions typed at the top level, if successfully type-checked, are assigned to the top-level "variable" `it`. This variable can be used in the next top level input to obtain the last value input.

```
-| 5;
val it = 5 : int

-| 2 + it;
val it = 7 : int
```

New variables with names of the user's choosing can be introduced using the `val` declaration. For example:

```
-| val xyz = 45 - 3;
val xyz = 42 : int
```

Note that the interactive loop prints the name of the variable declared, it's type, and its value.

2.1.2 Functions

MetaML is a higher order language. In it functions are *first class citizens*. They can be constructed, passed as parameters, and are recognized by the type-checker. For example

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

constructs an anonymous function which adds 1 to its argument. Using the normal interactive loop semantics, this anonymous function is then stored in the variable `it`. Note that functions are printed out by printing the function schema `fn`. Such functions can be applied to arguments to construct results.

```
-| fn x => x * x;
val it = fn : int -> int

-| (it 11);
val it = 121 : int
```

In order to write large programs, the normal SML programming style is to define functions, give them names, and then use these names in successive definitions .

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

-| val plus2 = fn x => inc(inc x);
val plus2 = fn : int -> int

-| plus2 4;
val it = 6 : int
```

MetaML supplies a useful syntactic sugar for defining functions.

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

-| (inc 12);
val it = 13 : int
```

This shorthand can be defined in terms of a named declaration (`val f = ...`) and an anonymous function definition (`fn => n+1`)[2]. The shorthand can be extended to functions of any arity. So we can use

- `fun f x = e` instead of `val f = fn x => e`.
- `fun g x y = r` instead of `val g = fn x => fn y => r` and
- `fun h x y z = p` instead of `val h = fn x => fn y => fn z => p`.

2.1.3 Polymorphism

SML also allows the programmer to write *polymorphic* functions. Such functions may be applied to arguments of different types. For instance the function `id` is the identity over values in any domain:

```
-| val id = fn x => x;
val id = Fn : ['a].'a -> 'a

-| id 3;
val it = 3 : int

-| id "a";
val it = "a" : string
```

[2]Strictly speaking, this is not exactly correct, as the `fun` syntactic sugar allows recursive functions while the `val` form does not. See Section 2.1.5

The type-checker indicates that `id` is polymorphic by using a type variable `'a` in its type. In MetaML, the list of the free[3] variables of a type are displayed as a prefix of the type: in the example below, `['a]` indicates that the type of the expression is `'a -> 'a`, for any `'a`.

Another example is the function `max` below. `max` takes three arguments: an operator `gt` for comparison and two values `a` and `b`. It returns the greatest element (according to `gt`) between `a` and `b`.

```
-| fun max gt x y = if (gt x y) then x else y;
val max = Fn : ['a].('a -> 'a -> bool) -> 'a -> 'a -> 'a
```

`max` is polymorphic. However, note that the system requires `x` and `y` have the same type `'a`. In addition `gt` has to have the type `'a -> 'a -> bool`.

2.1.4 Non-generalized type variables

Polymorphism allows the programmer to define generic functions which may be applied to different kinds of arguments. For instance the type of the function `id` (given in the previous section) is parameterized by a *type variable* `'a`:

```
-| fun id x = x;
val id = Fn : ['a].'a -> 'a
```

For instance, `id` may be applied to both strings and functions:

```
-| id "this is a string";
val it = "this is a string" : string

-| id (fn x => x+1);
val it = fn : int -> int
```

We say that `id`'s type variable is a *generalized* type variable. This indicates its polymorphic behavior. However, It may be sometimes unsafe to generalize a type variable. This "unsafeness" follows from an interaction of SML's reference types and generalized type variables. Generalization occurs at generalization points, typically at top-level and inside let bindings. At a generalization point, if an expression is not a value (but a computation which may allocate a reference cell), then it cannot be given a polymorphic type. For instance, let us define the function pair:

```
-| fun pair y = (y,y);
val pair = Fn : ['a].'a -> ('a * 'a)
```

Note that both `pair` and `id` are polymorphic The problem occurs if we consider the expression `(id pair)`. This expression is polymorphic but it is not a value (it is an application and it is feasible that such an application could allocate reference cells). Type safety considerations do not allow the generalization of expressions which do reference cell [Wri95] allocation.

[3] Some type variables are not free. See Section 2.1.4 for details.

The treatment of non-generalizable type variables varies, depending upon particular implementations. Standard SML of New Jersey's type-checker rejects such expressions. CAML [Ler97] instantiates these variables by *non-generalized type variables*. Any type may be assigned to a non-generalized type variable v. However, v may only be instantiated once. Like CAML METAML uses non-generalized type variables. The example below illustrates the behavior of the system concerning this point.

```
-|  val e = id pair;
val e = fn : '1 -> ('1 * '1)
```

The variable `e` has a type (`'1 -> ('1 * '1)`) which includes a non-generalizable type variable `'1`. Note the absence of square brackets, and the naming of the variable with a digit rather than an alphabetic character. This is METAML's way of indicating instantiatable type variables. This variable will become instantiated if `e` is ever used in a non-polymorphic way. Consider:

```
-| e 3;
val it = (3,3) : (int * int)
```

The application has instantiated the variable `'1`. The function `e` is now a function over integers. After this, attempting to apply `e` to an argument of a different type makes the system complain.

```
-| e;
val it = fn : int -> (int * int)

-| e "aa";
Error: The term:
"aa"
in file 'top level' 4 - 8
  Cannot unify the types: different type constructor
   string
  and
   int
 in expression: "aa"
```

2.1.5 Recursion

Recursion in SML is explicit. That is, if you want to write a recursive function you must use the `fun` syntactic sugar or the `val rec` declaration. To illustrate this, here are two different ways to express the factorial function recursively:

```
-| val rec fact = fn x => if x=0 then 1 else x*(fact (x-1));
val fact = fn : int -> int

-| fact 5;
val it = 120 : int
```

and

```
-| fun fact x = if x=0 then 1 else x*(fact (x-1));
val fact = fn : int -> int

-| fact 4;
val it = 24 : int
```

2.1.6 Data structures

New data-types are defined using the `datatype` construct.

```
-| datatype 'a Seq = N | C of 'a * 'a Seq;
constructor N
constructor C

-| N;
val it = Fn  : ['a].'a  Seq

-| C;
val it = Fn  : ['a].('a  * 'a  Seq) -> 'a  Seq
```

A *data-type declaration* introduces a new type (or in this case a type constructor) `Seq`, and several value constructors (`N` and `C`). A value constructor can either be a constant of the new type, such as `N` or a constructor function which when applied to values of the correct type *constructs* a value of the new type such as `C`.

```
-| val s = C(1,C(2,C(3,N)));
val s = C (1,C (2,C (3,N ))) : int Seq
```

As another example, a binary tree with integer values stored at the leaves could be defined as:

```
-| datatype Tree = Tip of int | Node of Tree * Tree;
constructor Tip
constructor Node

-| Tip;
val it = fn  : int -> Tree

-| Node;
val it = fn  : (Tree * Tree) -> Tree
```

Pattern matching allows the user to write functions over data types. `null` indicates whether a sequence as defined above by the data type `Seq` is empty. `length` computes the length of a sequence. The symbol _ matches any value.

```
-| fun length N = 0
     | length (C(x,xs)) = 1+ (length xs);
val length = Fn  : ['a].'a  Seq -> int
```

```
-| length s;
val it = 3 : int

-| fun null N = true
     | null (C _) = false;
val null = Fn  : ['a].'a  Seq -> bool

-| null s;
val it =  false  : bool
```

This completes our cursory review of Standard ML. We now turn our attention to the meta-level features unique to MetaML.

2.2 Multi-stage annotations

In an analogy with higher-order programming, a *second-order* program makes functions first class, while a *two-stage* program makes the *representations* of programs first class. A third-stage program makes representations of representations first class, etc. MetaML is a *multistage* programming system since it allows representations at all stages to be first class objects.

MetaML is designed to make an explicit distinction between the *syntax* and the *semantics* of programs. By this we mean that a distinction is made between the *value* described by an expression and the *encoding* of that expression. MetaML makes this distinction explicit and allows programmers to describe both.

The representation of a program `p` whose value is p is denoted by the meta-bracketed expression `<p>`. We say `<p>` is *stage*-1 *code* and denotes a representation of `p`. We refer to `p` as *stage*-0 *code* and this denotes its value p.

2.2.1 The bracket operator: building pieces of code

MetaML differs from SML in that it allows the programmer to explicitly manipulate *higher-stage expressions*[4]. In MetaML, a stage-1 expression is denoted by enclosing it between meta-brackets. For instance, the pieces of code denoting the constant 23 and the arithmetic expression $5+1$ are

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

-| <5+1>;
val it = <5 %+ 1> : <int>
```

First of all, let us remark that neither `<23>` nor `<5+1>` are integer values. The type of these two expressions (`<int>` and `<int>`) indicates that they are both pieces of code with type `int`.

Trying to use them as integers fails in the type checking stage, and the system complains:

```
-| <23> + 2;
Error: The term:
```

[4]Expressions whose stage is higher than 0.

```
(<23>,2)
in file 'top level' 2 - 10
  Cannot unify the types: Non Qualified type
   ?#
  and
   <int>
 in expression: (<23>,2)
```

Furthermore, `<5+1>` is not reduced to `6` because bracketed expressions are *frozen*. Both expressions are object programs which are not evaluated until explicitly forced with the `run` annotation.

A more sophisticated example contrasts the expressions `id` and `idCode` below:

```
-| val id = fn x => x;
val id = Fn : ['a].'a -> 'a

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

`id` is a pure SML function and the language infers its usual polymorphic type. Next, the type associated to `idCode` is `<'a -> 'a>`. Note the code type constructor brackets surrounding the function type. Code can be executed using the `run` construct.

```
-| (run idCode) "aa";
val it = "aa" : string
```

In the definition of `idCode`, `'a -> 'a` represents the type the object program will have, when finally executed.

In MetaML, programs (bracketed expressions) are first-class values. Hence one can use them in many different ways.

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

`<fn n => n + 1>` denotes the code of a function. It is the encoding of the increment function over integers.

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

`<fn n => <n + 1>>` denotes a three-stage program. In the first stage it is simply a piece of data which represents a program. In the second stage that program can be run, accepting an integer and producing a piece of code. At the third stage that piece of code can be executed.

The *stage* of any piece of code can be identified by the number of surrounding brackets. Simple values as `13` and `fn n => n+1` are stage-0 object programs. In `<fn n => n + 1>`, the function inside the brackets is stage-1 code. Finally, in `<fn n => <n + 1>>`, `n + 1` is a stage-2 piece of code.

2.2.2 The escape operator: composing pieces of code

Bracketed expressions can be viewed as *frozen*, i.e. reduction does not apply under brackets. However, is it 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 (˜) . For instance, let us examine the function `pair` below:

```
-| fun pair x = <(~x,~x)>;
val pair = Fn  : ['a].<'a > -> <('a  * 'a )>
```

The function `pair` takes a piece of code (of type `<'a>`)as input, and produces a new piece of code (of type `<('a * 'a)>`). It transforms the input code x into the code of the pair (x, x). To do this we must "splice" x in the resulting code in two places. This is done by escaping the occurrences of `x` in the definition of `pair`. In general, when inside brackets at stage 1, `~e` evaluates to e at stage 0 (hopefully returning a piece of code), and this piece of code is spliced into the bracketed expression at the point where the escaped expression occurred.

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

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

2.2.3 The `run` operator: reducing user constructed code

Here we explore two other operators provided by MetaML to manipulate pieces of code. The `run` operator is the explicit annotation used to indicate that it is now time to execute a delayed computation (i.e. an object program).

```
-| 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.

2.2.4 The `lift` operator: another way to build code

The lift operation does not modify the expressive power of MetaML. However, it is used for generating more readable programs (see Paragraph 2.2.5). 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.

```
-| <4+1>;
val it = <4 %+ 1> : <int>

-| lift (4+1);
val it = <5> : <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. This restriction is maintained by the type of `lift`.

```
-| lift;
val it = Fn  : ['a^].'a  -> <'a >

-| lift (fn x => x);
Error: The term:
(fn x => x)
in file 'top level' 7 - 8
  Cannot unify the types: Non Qualified type
   ?^
  and
   ? -> ?
 in expression: (fn x => x)
```

This type can be read as follows: For all `'a` which is liftable (the meaning of the carat (^)), `lift` has type `'a -> <'a >`. Note how an error occurs if lift is applied to a value with a functional type.

2.2.5 Lexical capture of free variables: constant pieces of code

Sometimes it is 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`, it is referenced at stage 1. Note that inside `codePair`, `n` is a free variable. 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 phenomena cross stage persistence. The pretty printer for code prints all cross stage persistent 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
```

To illustrate the difference between the two, let us examine one more example:

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

The function `liftLike` behaves like the operator `lift`, it evaluates its argument, and then returns a piece of code with that value. Because it operates by cross-stage persistence, any value may be so lifted, even function values. The difference between the operator `lift` and the this lexical capture of free variables is hidden inside of the implementation.

Cross-stage persistence constructs *efficient* code, but this code is often hard to inspect because we cannot *see* the value inside the constant in the resulting code. Contrast:

```
-| lift (3+4,9);
val it = <(7,9)> : <(int * int)>

-| liftLike (3+4,9);
val it = <%x> : <(int * int)>
```

2.2.6 Consistency of multi-stage programs

It is important that the programmer obey strict rules when escaping pieces of code to avoid what we refer to as *phase errors*. First of all, the stage of an escaped expression must always be non-negative. This implies that the number of brackets surrounding an expression e is always greater than the number of times e is escaped, and that there are no escaped expressions which are not inside brackets. Such errors are detected by the type-checker which prohibit the use of badly escaped expressions.

```
-| fn x => ~x;
Error: level-0 code cannot be escaped
```

Inside an escaped expression one must be careful not to use a variable which is bound at a higher stage than where it is being used. For example the code: `<fn x => ~x>` is obviously wrong. Here the variable `x` is used, while the body of the abstraction where it is bound is being constructed. The variable `x` cannot be interpreted at stage 0, because it is abstracted at stage 1. In summary, a piece of code containing stage-n objects cannot be escaped to a stage less than n. Such phase-errors are detected by the type-checker which prohibits their use.

```
-| <fn x => ~x>;
Error: The term:
x
in file 'top level' 13 - 14
variable bound in phase 1 used too early in phase 0
```

2.3 A simple two stage example

We describe here a two-stage version of the power function. This illustrates the use of the bracket and escape annotations. Our goal is to build a function which can be specialized w.r.t. a given exponent. First, we write the stage-0 `power` function with no annotations..

```
-| fun pow x n = if n=0 then 1 else x*(pow x (n-1));
val pow = fn : int -> int -> int
```

Assuming now, that the exponent n is known at stage-0 we can write a two-stage program `pow2` which will specialize `pow` on the value n. We do this by annotating (putting brackets and escapes on) the stage-0 program.

```
-| fun pow2 n =
       < fn x => ~(if n=0 then <1> else <x*(~(pow2 (n-1)) x)>) >;
val pow2 = fn  : int -> <int -> int>
```

`pow2` generates the encoding of a unary function (the outermost brackets). However, the body of this function depends upon the value of n. Since n is static (known at stage 0) we can escape the conditional and it can be reduced at program generation time. Both branches of the `if` construct a piece of code which will be spliced into the body of the function being generated. In the `then` branch this piece of code is the constant `<1>`. In the `else` branch, a piece of code is built, containing the multiplication of `x` by an expression which is spliced in. This spliced expression is built by a recursive call to `pow2`. Note that the computation of the escaped expression `(pow2 (n-1))` can be done at stage 0, since its argument `(n-1)` is completely known.

Applying an integer to `pow2` yields a specialized version of the function. Here, `(pow2 3)` is a program which no longer contains conditional nor recursive calls. Only multiplications remain to be performed.

```
-| (pow2 3);
val it = <(fn a => a %* a %* a %* 1)> : <int -> int>

-| (run it) 5;
val it = 125 : int
```

2.4 More-than-two-stage programs

In the previous sections, we only considered two stage programs. However, MetaML allows us to write programs including any level of staging, i.e. by using nested meta-bracketed expressions. We illustrate this by the function `asap` below. `asap` is a specialized version of the function `f` below:

```
-| fun f oper = (fn x => (fn y => (fn z => oper(x,oper(y,z)))));
val f = Fn  : ['a,'b].(('b  * 'a ) -> 'a ) -> 'b  -> 'b  -> 'a  -> 'a
```

`f` inputs a binary operator, three values `x`, `y` and `z` and computes `x oper (y oper z)`. We desire a multistage version of `f` which specializes itself as soon as possible, i.e. as soon as `x`, `y` or `z` is known. We call this function `asap`. Note that this example is not completely general: we could use more than three variables and two different operators `oper1` and `oper2` instead of using `oper` twice. The annotated function is:

```
-| fun asap oper =
      <fn x =>
          <fn y =>
              <fn z => oper( ~(lift (oper(~(lift x),y))), z) >>>;
val asap = Fn  : ['a,'b^].
                 (('b * 'a ) -> 'b ) -> <'b  -> <'a  -> <'a  -> 'b >>>
```

This three-stage function creates a piece of code depending on a static value `oper`. As soon as `x` is known, we can generate the code of a new function which only depends on `y` and `z`, and so on for the other arguments:

```
-| asap (op +);
val it =
<(fn a => <(fn b => <(fn c => %oper (~%lift (%oper (~%lift a,b)),c))>)>)>
 : <'1  -> <'1  -> <'1  -> '1 >>>

-| (run it) 11;
val it =
<(fn a => <(fn b => %oper (~%lift (%oper (11,a)),b))>)>
 : <int -> <int -> int>>

-| (run it) 22;
val it = <(fn a => %oper (44,a))> : <int -> int>

-| (run it) 33;
val it = 77 : int
```

Note that the use of `lift` allows the actual execution of `oper(x,y)` as soon as `x` and `y` are known. If we had used a an escape (~) instead, this operation would not be executed until the resulting piece of code is run.

3 Language conventions and differences from SML

Since METAML is not Standard ML, there are a few places where the syntax and semantics of METAML differs from SML. In this section we summarize these differences.

3.1 The syntax of meta-constructors

Here we examine the few differences between SML's and MetaML's syntaxes. First of all, the syntax is extended to support the operators for meta-programs description.

- <e> freezes the expression e. Let $n \geq 0$ be the current stage of computation. <e> denotes a stage-$n+1$ expression. The computation of e is deferred, and <e> is treated as a value at stage n.
- ~e escapes from a frozen expression. Let ~e occur in a larger expression e'. Let $n > 0$ be the stage of e'. ~e is treated as stage-$(n-1)$ code, even if it is inside a stage-n expression.
- run e allows one to execute a piece of code e built at the same stage. Let e be a stage-n piece of code built at stage m, $n > m$. run e permits the computation of e at stage m. The resulting expression is a stage-$(m-1)$ expression.
- lift e converts *first-order expression* into a piece of code. However, it differs from the meta-brackets in that the expression is evaluated first. Note that this operator cannot be applied to functions.

3.2 The precedence of meta-constructors

The following describes the precedence rules used when parsing the meta-programming constructs.

1. Escape (~) has the highest precedence. Its priority is even higher than that of function application.
2. run has the lowest precedence in the language. Because of this, its scope extends as far to the right as possible.

For instance, <f ~x y> is parsed as <f (~x) y>, instead of <f ~(x y)> because of escape's high priority.

3.3 Disambiguating relational operators and meta-brackets

Finally, since the symbols < and > describe frozen expressions, we need a non-ambiguous way for writing comparison operators over numbers. MetaML uses the symbols '<' and '>' instead of SML < and > notations. The example below describes the use of the greater operator.

```
-| let val x = 1 in x '>' 0 end;
val it =  true  : bool
```

3.4 Lack of Module Level Features

MetaML is an implementation of the core of ML. It lacks all of ML's module features. This includes signatures, structures, functors, and qualified names such as Int.toString. MetaML does implement the abstype declaration, which is not strictly part of the module system.

```
abstype Complex = D of real * real
  with
    fun toComplex x = (D(x,0.0));
    fun realPart (D(x,y)) = x;
    fun imagPart (D(x,y)) = y;
    fun Add (D(x,y)) (D(a,b)) = D(x+a,y+b);
    end;
```

3.5 Qualified Types

In Standard ML, equality is handled in an ad-hoc fashion. The equality operator (`=`), has a polymorphic type `all ''t . ''t -> ''t -> bool` The notation `''t` is called an equality type variable, and is a generic type variable with restrictions. These restrictions ensure that an equality type variable can never be instantiated on a type which does not support equality.

This restriction is a special case of a more general mechanism called qualified types, which has found wide use in the Haskell language. In Haskell programmers can use the class system, which introduces qualified types, to specify over-loaded functions. The class system allows programmers to specify exactly which types can be used to instantiated qualified type variables, and which functions should be used at different types.

In MetaML we borrow a simple subset of the qualified type system of Haskell. In MetaML there are a fixed number of type qualifications. This set cannot be extended by the programmer, and all overloading is handled inside the interpreter, and is not user extendible.

In MetaML we write qualifications by postfixing a type variable by its qualification. Thus `'a=` is qualified by the *equality* predicate, `'b\name` is qualified by the *lacks the label* `name` predicate, `'c^` is qualified by the *liftable* predicate, `'d#` is qualified by the *numeric* predicate, and `'e<` is qualified by the `orderable` predicate.

The equality predicate applies to all types which do not have a function type inside. These are the observables of the language. The nice property of qualified types is that the predicates are propogated by the type system. Note below how the equality predicate is propagated into the types of the functions when they use the equality operator.

```
-| fun mem x [] = false | mem x (y::ys) = if x=y then false else mem x ys;
val mem = Fn  : ['a=].'a  -> 'a  list -> bool

-| fun find x ((y,z)::ys) = if x=y then z else find x ys;
val find = Fn  : ['a,'b=].'b  -> ('b  * 'a ) list -> 'a

-| fun f x y = (x,3) = y;
val f = Fn  : ['a=].'a  -> ('a  * int) -> bool
```

The liftable predicate applies to all observable types (like the equality predicate, types without functions) except code types. We do not allow the lifting of code types.

```
-| fun f x = lift x;
val f = Fn  : ['a^].'a  -> <'a >
```

The numeric predicate applies to `int` and `float`, and is manifest in the arithmetic operators.

```
-| fun f x = x + x;
val f = Fn  : ['a#].'a  -> 'a
```

The orderable predicate applies to `int`, `float`, `char`, and `string`, and is manifest in the relational operators.

```
-| fun f x = x '<' x;
val f = Fn  : ['a<].'a  -> bool
```

The lacks label predicate is used to type labled sums , and products (records). It applies to sum and products which lack the appropriate label.

```
-| fun f x = #name x;
val f = Fn  : ['a\name,'b].{name: 'b  | 'a } -> 'b
```

A type variable can have multiple qualifications. Consider the example below where `'a` must lack two labels.

```
-| fun f x = (#name x, #age x);
val f = Fn : ['a\name\age,'b,'c].{age: 'b ,name: 'c | 'a } -> ('c * 'b)
```

Or the example where the argument must meet both the numeric and orderable predicates.

```
-|  fun f x = ( x=x, x '<' x);
val f = Fn  : ['a=<].'a  -> (bool * bool)
```

3.6 Use v.s. The Interactive Loop

There are two ways that users can load functions into MetaML. Users can type functions directly into the top-level interactive loop, or they can use the file-oriented `use` function.

```
-| use "lists.sml";
val not = fn  : bool -> bool
val null = Fn  : ['a].'a  list -> bool
val hd = Fn  : ['a].'a  list -> 'a
val tl = Fn  : ['a].'a  list -> 'a  list
val length = Fn  : ['a].'a  list -> int
val append = Fn  : ['a].'a  list -> 'a  list -> 'a  list
val @ = Fn  : ['a].('a  list * 'a  list) -> 'a  list
val revappend = Fn  : ['a].('a  list * 'a  list) -> 'a  list
val rev = Fn  : ['a].'a  list -> 'a  list
val map = Fn  : ['a,'b].('b  -> 'a ) -> 'b  list -> 'a  list
```

By placing defintions in a file, and reading them with the `use` function programmers can save repeatedly typing the same definitions.

There is one important difference between the interactive loop and the `use` function. The parsers for these two modes are slightly different. When parsing a file, the parser can determine the end of the file by reading the end-of-file marker. When reading from the interactive loop, this is not possible. Because of this, the interactive loop determines the end of the current input when it senses a semicolon as the last character in a line. Problems are possible using this solution since, legal declarations in ML do not need to end in a semicolon, and semicolons can appear at the end of a line, before the whole declaration is parsed. For example:

```
-| let val x = 12;
in x + 1 end;
Error: 2.0: syntax error found at EOF

-| Error: 1.2: syntax error: deleting  IN ID
Error: 2.11: syntax error found at END
```

The `let` is a legal ML declaration but the semicolon after the `12` prematurely ends the parse, and the `in x + 1 end` fragment is taken as the beginning of the next interaction. Had this appeared in a file parsed by `use` this error would not have occured.

4 Typing Rules

The meta-operators add a certain level of complexity to the typing rules of SML. In this section we treat this additional complexity.

4.1 The purpose of a multi-stage type system

A formal treatment of the static semantics of multistage programs can be found in [TBS98, MTBS99, She97, Tah99]. Here we informally describe how higher-stage expressions are checked by the system. MetaML's type-checker helps the programmer by verifying the consistency of multistage expressions. This is three-fold:

1. A particular *code type* is assigned to every higher-stage expression. This code-type includes information completely describing the type of the delayed program contained. The semantics of MetaML assumes that every piece of code must execute safely in an environment with no bindings for free variables. Most free variables are converted into cross stage persistent constants, but the interaction between meta-brackets and the `run` operator allows a few free variables to escape this mechanism. For example, `<fn x =>  (run <x>)>` attempts to "lookup" `x` in the empty environment. This interaction between run and escape is the one hole in the current MetaML implementation. See section **??** for a discussion of this.

2. The system checks the consistency of higher-stage expressions. By this we mean that any delayed program, is itself always well typed. For instance the expression `<1+true>` is rejected even if it could be considered as a piece of code at stage 0. This means that any well-typed expression cannot go wrong at any stage of its execution.

3. The phase consistency of multistage expressions is verified by the type-checker. This prevents the programmer from writing programs which use variables in stages before they are defined. For instance it prevents the use of an expression in a stage preceding its computation such as `<fn x => ~x>`, but does not catch the error caused by the interaction between run and escape.

4.2 Typing Multi-stage expressions

MetaML assigns types of the form <α> to every meta-bracketed expression. The brackets surrounding this pair indicate that it is a stage-1 expression. Level-n *expressions* are surrounded by n pairs of brackets, and stage-n *types* are also surrounded by n pairs of brackets. A stage-0 expression is surrounded by zero pairs of brackets: stage-0 expressions are assigned normal types. This is illustrated in the examples below.

```
-| <2>;
val it = <2> : <int>
```

Here we have a stage-1 expression where $\alpha =$ int. In the type <α>, α indicates the type the expression will have in the stage of its execution. Obviously, in <2>, 2 will have type int at stage-1.

```
-| <<<2>>>;
val it = <<<2>>> : <<<int>>>
```

Here we have a stage-3 expression. $\alpha =$ is itself a code type. We can also have code types for functions.

```
-| <fn x => x+5>;
val it = <(fn a => a %+ 5)> : <int -> int>

-| fn x => <x+5>;
val it = fn  : int -> <int>
```

Summarizing the type rules for the meta-programming constructs we have:

- **meta-brackets.** If e has type τ then <e> has type <τ>. In addition any escapes (or "holes") inside of e must be filled with code of the correct type for that holes context.
- **escape.** If e has type <τ> then ~e has type τ and can be spliced into a "hole" whose context expects a piece of code with type <τ>.
- **run.** If e has type <τ> then run e has type τ.
- **lift.** If e has type τ then lift e has type <τ>

4.3 Interaction Between run and escape

??

The type system presented above has some drawbacks. In particular it types the program <fn x => ~(run <x>)> which leads to a runtime error.

```
-|  <fn x => ~(run <x>)>;;
Error: In the dynamic runtime environment: Variable Not Found: 'x'782'
```

This is because the rule for typing `run` expressions 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[TBS98, MTBS99]. Unfortunately these systems either also throw away other good programs, or require elaborate annotations. A complete solution to this problem can be found in the thesis of Walid Taha[Tah99], but this requires writing programs in a rather unnatural style.

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.

5 Generation time code optimizations

To write multistage programs effectively, one needs to observe the programs produced, and these programs should be as simple as possible. For this reason, our implementation performs automatic optimizations on several kinds of code when it is generated. These optimizations involve performing safe beta and eta reductions, let-hoisting and monad law normalization.

5.1 Safe Beta Reduction

A beta reduction is safe if it does not change evaluation order, or effect termination properties. There is one safe case which is particularly easy to recognize, namely, Plotkin's β_v rule [Plo75]. 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 symbolically beta reduced. In order to avoid duplicating code we restrict our optimizations to constants or variables (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>`.

5.2 Safe Eta reduction

Another simple optimization 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)> : ['a,'b].<('b  -> 'a ) -> 'b  -> 'a >

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

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.

5.3 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 [SW97]. 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.

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
```

6 The art of multi-stage programming

In this Section we introduce two larger examples of two-stage programs. The first example is a multistage function for merging sorted lists. The second example is a program which generates specialized versions of the shortest path algorithm specialized for a particular graph. These examples are presented, to point out in more detail, how MetaML works, and tom illustrate some of the problems that need to be addressed when staging and algorithm.

6.1 Merging lists

In this Section, we examine how to annotate `merge`, a simple function over sorted lists.

6.1.1 The standard `merge` function

Given two sorted sequences `s1` and `s2`, `merge` outputs the sorted concatenation of `s1` and `s2`. We use this example in order to illustrate the way meta-programs can be designed. One standard-SML `merge` function is

```
-| fun merge xs ys =
   case xs of
     [] => ys
   | (z::zs) => case ys of
                  [] => xs
                | w::ws => if z '<' (w:int)
                           then z::(merge zs ys)
                           else w::(merge xs ws);
val merge = fn  : int list -> int list -> int list
```

The example below illustrates how the two sorted sequences `s1` and `s2` can be used as arguments to `merge` to produce a single sorted sequence.

```
-|  val s1 = [3,6,7];
val s1 = [3,6,7] : int list

-|  val s2 = [2,9,43];
val s2 = [2,9,43] : int list

-|  (merge s1 s2);
val it = [2,3,6,7,9,43] : int list
```

6.1.2 Annotating `merge`: a first attempt

Let us assume that only sequence `s1` is known. Our goal is to specialize `merge` with respect to this static parameter. Straightforward annotations of `merge` yield the function `merge2`.

```
-| fun merge2 xs =
   <fn ys =>
     ~(case xs of
         [] => <ys>
      | (z::zs) => <case ys of
                      [] => ~(lift xs)
                    | (w::ws) => if z '<' (w:int)
                                 then z::(~(merge2 zs) ys)
                                 else w::(~(merge2 xs) ws)
                   >
      )
   >;
val merge2 = fn  : int list -> <int list -> int list>
```

merge2 is a function which inputs a sequence xs and outputs a program p. First, p abstracts the sequence ys to be merged into xs. The escape character, around the case construct, makes the case level-0 code. The case analysis of xs is performed in the first stage and the result is a piece of code (a level-1 expression). When xs matches z::zs, the recursive calls to merge2 are escaped. This is possible because xs is known at level 0.

The type of merge2 indicates that it is a function which inputs a sequence of integers and generates the encoding of a function from the domain of sequences of integers to a range of sequence of integers.

However, applying merge2 to s1 yields an infinite computation. This is because the computation of w::(~(merge2 xs) ws) in the else branch contains a non-decreasing recursive call to merge2.

This obvious annotation of merge produces an incorrect program. A more careful examination of the function is required for multistage programming.

6.1.3 Annotating merge: a second attempt

In order to ensure termination, the recursive call to the function in the else statement must be changed. merge3 is a non annotated version of merge which restructures the recursion pattern of merge. The function split z ys returns a pair of sequences containing the elements in ys greater than and less than z respectively. The head of the resulting sequence is u (the elements less than z), followed by z. Now all recursive calls are strictly decreasing. merge3 uses the auxiliary function split below.

```
-| fun reverse [] ys = ys | reverse (x::xs) ys = (reverse xs (x::ys));
val reverse = Fn  : ['a].'a  list -> 'a  list -> 'a  list

-| fun split2 (n:int) xs (small,big) =
     case xs of
       [] => (reverse small [], reverse big [])
     | (z::zs) => if z '>' n
                    then split2 n zs (small,z::big)
                    else split2 n zs (z::small,big);
val split2 = fn
    : int -> int list  -> (int list  * int list ) -> (int list  * int list )

-| fun split n l = (split2 n l ([],[]));
val split = fn : int -> int list  -> (int list  * int list )

-| fun merge3 xs ys =
   case xs of
     [] => ys
   | z::zs =>
       (case ys of
          [] => xs
        | w::ws =>
           if z '<' w
              then z::(merge3 zs ys)
              else let val (u,v) = split z ys
```

```
                   in
                     append u (z::(merge3 zs v))
                   end
       );
val merge3 = fn : int list  -> int list  -> int list
```

merge4 is an annotated version of merge3. Annotations in merge4 are very similar to those in merge2. However, the termination behavior of the first stage of the computation has now been improved.

```
-| fun merge4 xs =
<fn ys =>
  ~(case xs of
      [] => <ys>
    | (z::zs) =>
        <case ys of
           [] => ~(lift xs)
         | (w::ws) =>
             if (~(lift z)) '<' w
                then (~(lift z))::(~(merge4 zs) ys)
                else let val (low,high) = split (~(lift z)) ys
                     in
                       append low
                              (~(lift z)::(~(merge4 zs) high))
                     end
        >)
>;
val merge4 = fn  : int list -> <int list -> int list>
```

It is important to note that split defined at level 0 is used at level 1 in the definition of merge4. The importance of cross stage persistence when writing staged programs cannot be over emphasized. The example below depicts the application of a one element list to merge4. The specialized program inputs a list and inserts the static 2 in the right place.

```
-|  (merge4 [2]);

val it =
<(fn a =>
     (case a of
       [] =>
          [2]
     | (c::b) =>
          if 2 %'<' c
             then 2 :: a
             else let val (e,d) = %split 2 a  in %append e (2 :: d) end))>
 : <int list -> int list>
```

6.1.4 Decreasing the size of the output program

Although merge4 is correct, the generated program has size exponential in the length of the static input sequence. This is because all the branches induced by the conditional `if z '<' w` are explored.

Another approach consists of using a different definition of the merge function, denoted merge5.

```
-| fun merge5 xs ys =
   case xs of
     [] => ys
   | (a::b) => let val (u,v) = split a ys
               in
                 append u (a::(merge5 b v))
               end;
val merge5 = fn : int list  -> int list  -> int list
```

The `if` statement occurring in merge3 and merge4 has disappeared in merge5. Concisely, its purpose has been *pushed* inside the definition of split which will be executed at level-1. merge6 is the natural annotated version of merge5.

```
-| fun merge6 xs =
   <fn ys =>
     ~(case xs of
         [] => <ys>
       | (a::b) => <let val (u,v) = split ~(lift a) ys
                    in
                      append u (~(lift a)::(~(merge6 b) v))
                    end
                   >
      )
   >;
val merge6 = fn  : int list -> <int list -> int list>
```

The program bellow was generated using merge6. Its size is linear in this of the input sequence.

```
-| merge6 [2];
val it =
<(fn a => let val (c,b) = %split 2 a  in %append c (2 :: b) end)>
 : <int list -> int list>
```

When applied to larger static lists, the increase in size of the generated program is linear with the size of the input. It adds a single `let` for each element.

```
-| merge6 [2,5];
val it =
<(fn a =>
     let val (c,b) = %split 2 a
      in %append c (2 :: let val (e,d) = %split 5 b  in %append e (5 :: d) end) end)>
 : <int list -> int list>
```

The appearance of this generated program could be improved by "pulling" the let outside of the cons (::) operator.

```
<fn a => let val (c,b) = %split 2 a
             val (e,d) = %split 5 b
             val w = (%append) e (2  ::  d)
         in (%append) c (5  :: w)
         end>
```

This can be done by restructuring the generator. To effect this transformation, we introduce a new local variable `zs` in the generated (second stage) code, and assign to it the expression generated by the (first stage) recursive call to `merge6`.

```
fun merge6 xs =
   <fn ys =>
     ~(case xs of
         [] => <ys>
       | (a::b) => <let val (u,v) = split ~(lift a) ys
                            val zs = ~(merge6 b) v
                        in
                          append u (~(lift a):: zs)
                        end
                       >
      )
   >;
```

Now observe the structure of generated code for a larger first stage input.

```
-| merge6 [1,5,8,33];
val it =
<(fn a =>
     let val (c,b) = %split 1 a
         val (e,d) = %split 5 b
         val (g,f) = %split 8 d
         val (i,h) = %split 33 f
         val j = h
         val k = %append i (33 :: j)
         val l = %append g (8 :: k)
         val m = %append e (5 :: l)
      in %append c (1 :: m) end)>
 : <int list -> int list>
```

6.2 Specializing a shortest-path algorithm

This Section illustrates how multistage programming can be used to efficiently improve a program that solves a problem in graph theory. Given a directed weighted graph (V, E) and two nodes $a, b \in V$, a shortest path algorithm finds the minimum weighted path in G from the source a to the destination b.

Assume now that the weight function is unknown at program generation-time. One can expect to specialize the general algorithm with respect to the particular graph G. In the remainder of this Section, we present a multistage program that realizes such a specialization.

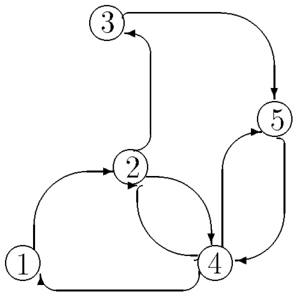

Figure 1: The graph represented by the function `g`.

6.2.1 Graph representation

A graph is encoded by a function `g`. The function `g`'s argument can be thought of as a vertex, denoted by an integer label. The function returns a list of successor nodes denoted by their integer labels. The function `g` below encodes the graph in Figure 1.

```
-| fun g x =
     case x of
       1 => [2]
     | 2 => [3,4]
     | 3 => [5]
     | 4 => [1,2,5]
     | 5 => [4]
     | _ => [];
val g = fn  : int -> int list
```

6.2.2 The shortest path algorithm

First we describe the standard algorithm. `shortestPath` is a non-annotated function, taking five arguments.

- `succ` is the function associating to a vertex a list of successor vertexes, and encodes the topology of the graph;
- `source` is the source vertex;
- `dest` is the destination vertex;
- `marked` indicates which nodes have already been visited by the algorithm;
- `weight x y` indicates the weight of the edge (x, y).

The resulting value is an integer indicating the cost of the shortest path between `source` and `dest` in the graph denoted by `succ`. In the `then` branch of the conditional, 10000 is assumed to be an infinitely big value. First, `source` is added to the list of already visited vertices. `explore` is the list of `source`'s successors which have not been visited yet. The function inside the `map` computes the shortest path between each of `source's` successors and `dest`. The resulting values are stored in the `pathWeights` list. Finally, `shortestPath` returns the minimum value in `pathWeights`.

```
-| fun shortestPath succ source dest marked weight =
     if source = dest
        then 0
        else
          let val marked2 = (source::marked)
              val explore = (minus (succ source) marked2)
              fun short x = shortestPath succ x dest marked2 weight
              val path_weights =
                   map (fn node => (short node) + (weight source node))
                       explore
          in
            case path_weights of
              [] => 10000
            | _ => (minimum path_weights)
          end;
val shortestPath = fn
  : (int->int list) -> int -> int -> int list  -> (int->int->int) -> int
```

The following example illustrates the use of the function (here every edge has a unit weight, i.e. `w x y = 1`).

```
-| (shortestPath g 3 2 [] w);
val it = 3 : int
```

6.2.3 A two-level shortest path algorithm

Here, we describe how to annotate the algorithm in order to obtain a program generator which produces graph-dependent code. Since the `weight` function is assumed unknown, `shortestPath2` outputs the encoding of a function depending on this parameter. Most of the computation only depends on the structure of the graph and can be escaped. Hence, the variables `marked2` and `explore` are bound at program generation-time. In addition, inside the `map function`, only the application of `weight` and the addition are not performed statically.

```
-| fun shortestPath2 succ source dest marked =
   <fn weight =>
     ~(if source = dest
          then <0>
          else
            let val marked2 = source::marked
                val explore = (minus (succ source) marked)
                fun short x = shortestPath2 succ x dest marked2
                val path_weights =
                      map (fn node => < (~(short node) weight) +
                             (weight ~(lift source) ~(lift node)) > )
                          explore
```

```
            in
               (liftMinimum path_weights)
            end
      )
   >;
val shortestPath2 = fn
 : (int -> int list ) -> int -> int -> int list  ->
                            <(int -> int -> int) -> int>
```

Finally, note that the minimum function has to generate a residual code. This is done by lifting it into the new function `liftMinimum` defined below.

```
-| fun liftMinimum x =
     case x of
       [] => <10000>
     | (c::cs) => <min ~c ~(liftMinimum cs)>;
val liftMinimum = fn  : <int> list -> <int>
```

The following example illustrate the use of `shortestPath2`. We obtain very compact code.

```
-| (shortestPath2 g 4 2 []);
<(fn weight =>
     %min (%min (0 %+ weight 1 2) 10000 %+ weight 4 1)
          (%min (0 %+ weight 4 2) (%min (10000 %+ weight 4 5) 10000)))>
 : <(int -> int -> int) -> int>
```

It is possible to improve the appearance of this residual program by restructuring the `shortest_path` and the `liftMinimum` function. By introducing local `let` bindings we avoid the deeply nested calls to the `min` function.

```
-| fun liftMinimum x =
     case x of
       [] => <10000>
     | ( <10000> :: cs) => liftMinimum cs
     | (c::cs) => <let val y = ~(liftMinimum cs)
                       val z = ~c
                   in min z y end>;
val liftMinimum = fn  : <int> list -> <int>

-| fun shortestPath2 succ source dest marked =
   <fn weight =>
     ~(if source = dest
          then <0>
          else
            let val marked2 = source::marked
                val explore = (minus (succ source) marked)
                fun short x = shortestPath2 succ x dest marked2
                val path_weights =
                      map (fn node =>
```

```
                              let val recall = < ~(short node) weight>
                                  val wght = <weight ~(lift source) ~(lift node)>
                              in <let val q = ~recall in q + ~wght end>
                              end)
                          explore
          in
             (liftMinimum path_weights)
          end
      )
   >;
val shortestPath2 = fn
 : (int -> int list ) -> int -> int -> int list  -> <(int -> int -> int) -> int>

-|  (shortestPath2 g 4 2 []);
-| val it =
<(fn a =>
     let val b = 10000
         val c = 10000
         val d = c %+ a 4 5
         val e = %min d b
         val f = 0
         val g = f %+ a 4 2
         val h = %min g e
         val i = 10000
         val j = 0
         val k = j %+ a 1 2
         val l = %min k i
         val m = l %+ a 4 1
      in %min m h end)>
 : <(int -> int -> int) -> int>
```

For completeness we include the definitions of the auxiliary functions used by `shortestPath` and `shortestPath2` below.

```
(*  min : int -> int -> int *)
fun min x z = if x '<' z then x else z ;

(* minimum : int list  -> int *)
fun minimum (x::xs) = (case xs of [] => x | _ => min x (minimum xs));

(* map : ('b -> 'a) -> 'b list  -> 'a list *)
fun map f x = case x of [] => [] | (a::b) => (f a)::(map f b);

(* mem : int -> int list  -> bool *)
fun mem x xs = case xs of [] => false | (a::b) => if a=x then true else mem x b;

(* minus : int list  -> int list  -> int list *)
fun minus xs ys =
```

```
case xs of
  [] => []
| (a::b) => if mem a ys then minus b ys else a::(minus b ys);
```

7 Extensions to Standard ML

We built MetaML to experiment with staged programming. We found a number of applications (especially in the area of compiler generation from staged interpreters) required flexible type systems that were not available in Standard ML. Thus we implemented a number of extensions to Standard ML in MetaML. These extensions are sound and well thought out ideas which have appeared in the literature. There are no new ideas in these extensions, only practical implementations. These extensions allow a broader style of programming, and an implementation of them may be useful to programmers in their own right, even if their problems do not involve staging.

7.1 Flexible Records

MetaML uses a more general notion for record types than Standard ML. A record is a labeled product. A record contains any number of fields surrounded by squiggly brackets (`{ ... }`). A field is composed of a *label* and a value separated by an equal sign (`=`). A record with two fields called `name` and `age` can be written: { `name = "Tom", age = 29` }. Each field has an associated type and the type of a record is written as sequence of labels separated by a colon (`:`). Thus the record above has type: { `name : string, age : int` }.

Field selection is accomplished by pattern matching or the record selector (`#`) notation. Thus `#name` { `name = "Tom", age = 29` } reduces to `"Tom"`. A record selector is written by using a hash followed by an integer, or alpha-numeric token. The types of selectors are quite generic, and apply to any record with a label the same as the selector. For example, we may observe] the type of the `#name` selector by typing it at the MetaML prompt:

```
-| #name;
val it = Fn  : ['a\name,'b].{name: 'b  | 'a } -> 'b
```

We read this type as follows: for all records of type `'a` which don't have a label called `name`, and all types `'b`, the expression `#name` has a function type from a record with all the fields that `'a` has plus one more field with label `name` of type `'b` to a `'b` object.

The notation `'a\name` is called a qualified type. It means all types which are records, and which additionally do not have a field with label `name`.

One may also use pattern matching to access the fields of a record. For example consider:

```
-| fun g {name=x, age = w, ... } = w+1;
val g = Fn  : ['a\age\name,'b].{age: int,name: 'b  | 'a } -> int
```

Record extension is the ability to construct a new record with additional fields from an old record. The record extension notation adds a vertical bar (`|`) to the squiggly bracket record notation.

```
-| fun j {name=x, age = w | z } = (w+1,{name =  "Joe" |z});
val j = Fn  : ['a\age\name,'b].
              {age: int,name: 'b  | 'a } ->
              (int * {name: string | 'a })

-| j {name = "Tom" , age = 29, address = "1 Main Street"};
val it = (30,{address = "1 Main Street", name = "Joe"}) :
         (int * {address: string,name: string | '1 })
```

7.2 Labelled Anonymous Sums

Labelled anonymous sums are the dual to labeled products. Here the label is an injection tag into a sum type. In analogy to records, injections in MetaML are indicated using the percent (`%`) notation. A percent followed by an integer, or alpha-numeric token indicates an injection.

```
-| %even;
val it = Fn  : ['a\even,'b].'b  -> Sum{even: 'b  | 'a }
```

When applied to arguments, injections produce labeled sum types.

```
-| %even 4;
val it = %even 4 : Sum{even: int | '1 }

-| %A true;
val it = %A true  : Sum{A: bool | '1 }
```

The nice property of labeled sum types is that sumtypes unify to construct richer sum types. Since both branches of the if in the function below return a different sum type, the range of the function is the union of the two sum types.

```
-| fun f x = if even x then %even x else %odd x;
val f = Fn  : ['a\even\odd].int -> Sum{even: int,odd: int | 'a }
```

Sum types can be discriminated using pattern matching. The pattern `( %name x )` matches only against a variant of a sum type whose actual injection tag is `%name`. Observe the interaction below:

```
-| fun getA (%a x) = x;
val getA = Fn  : ['a\a,'b].Sum{a: 'b  | 'a } -> 'b

-| getA (%a 3);
val it = 3 : int

-| getA( %B false );
Non Exhaustive Match
%B false
```

A multi-clause function (`(fn x => clause1 | y => clause2)`) with injections as patterns, is typed by removing the label of each clause from the sum type of additional clauses.

For example the type of `z` in the function below is a sum type (`'a\w`) without a `w` injection.

```
-| fun W g =  (fn %w x => g x | z => z);
val W = Fn  : ['a\w,'b].('b  -> 'a ) -> Sum{w: 'b  | 'a } -> 'a
```

7.3 Polymorphic Recursion

In MetaML it is possible to type functions which use polymorphic recursion. Polymorphic recursion occurs when in a recursive function definition, the function being defined is called at two different instances of a polymorphic type. In general Hindley-Milner type inference cannot be used to infer the type of a function that uses polymorphic recursion.

For example consider the datatype below, and our first attempt at writing the the map function for it.

```
-| datatype 'a T = A1 of 'a | A2 of 'a T * 'a T | A3 of ('a list) T;
constructor A1
constructor A2
constructor A3

-| fun mapT f (A1 x) = f x
     | mapT f (A2(x,y)) = A2(mapT f x,mapT f y)
     | mapT f (A3 zs) = A3(map f zs);
Error: The term:
map
in file 'top level' 101 - 104
  Cannot unify the types: different type constructor
   list
  and
   T
 in expression: map
```

By giving an explicit type declaration for the function we can at least check that the given type is well formed. To give a type declaration in MetaML it is necessary to use the `val rec` form of recursive function definition (this is a restriction of the current implementation, and will hopefully be removed in the future).

```
-| val rec ('a,'b,'c) mapT : ('a -> 'b) -> 'a T -> 'b T =
fn f => (fn (A1 x) => A1(f x)
          | (A2(a,b)) => A2(mapT f a, mapT f b)
          | (A3 w) => A3(m (map f) w));
val m = Fn  : ['a,'b].('b  -> 'a ) -> 'b  T -> 'a  T
```

7.4 Local (non Hindley-Milner) Polymorphism

MetaML allows a certain level of local polymorphism. Most languages with type inference, allow universal quantification only at the outermost level of a type. This makes it impossible to write a higher order function that takes a polymorphic function as an argument. Such a function would have a type like: `all 'b .  (all 'a .  'a T -> 'b S) -> 'b -> int` where some of the universally quantified type variables (the `'a` in this example) are not at the outermost level (like the `'b`).

MetaML relaxes this restriction by allowing the arguments to data constructor functions to be locally universally quantified. For example we define the constructor `A` to take two parameters both of which are polymorphic functions.

```
-| datatype Z = A of (['a].'a -> 'a) *(['b].'b -> int);
constructor A
```

We can apply A only to suitably polymorphic arguments. For example:

```
-| fun id x = x;
val id = Fn  : ['a].'a  -> 'a

-| fun two x = 2;
val two = Fn  : ['a].'a  -> int

-| val q = A(id,two);
val q = A (Fn ,Fn ) : Z
```

Note that the type of q is Z which does not have any polymorphism in its type. Thus we can get around the restriction that no function can be passed a polymorphic function as an argument, by passing it a Z-like object instead which suitably encapsulates polymorphic functions.

Now a function that takes a object A(h,k) with type Z as a parameter can use its polymorphic components in a polymorphic way.

```
-| fun g (A(h,k)) = (h "a",h 4,k 2, k true);
val g = fn  : Z -> (string * int * int * int)
```

In order for this to be safe, some fairly strict restrictions must be placed on the arguments to constructors that expect polymorphic components. First, the components must really be polymorphic, in that their types make no mention of type variables bound in an outer scope.

```
-| fn f => A(id,f);
Error: The term:
A(id,f)
in file 'top level' 10 - 17
Argument of polymorphic constructor has type which binds its
polymorphic type in outer scope  'a -> int
```

Or that they bound to some type at all, no longer being polymorphic.

```
-| A(id,id);
Error: The term:
A(id,id)
in file 'top level' 2 - 10
Polymorphic type variable in argument to polymorphic function gets
bound to: int
```

7.5 Local Existential Quantification

MetaML also allows the arguments to data constructors to be existentially quantified. This allows the construction of first class abstract datatypes. Local existentially quantified values use the syntax below:

```
-| datatype 'a T = C of (exists['b].('b * ('b -> 'a)));
constructor C
```

This defines a new constructor `C` which takes a pair as an argument. The pair consists of a value of type `'b` and a function of type `'b -> 'a`, where the type `'b` is hidden from the user. Let us construct several such values and define a function that uses the components encapsulated by the `C` constructor function:

```
-| val w1 = C(5,fn x => x + 1);
val w1 = C Fn  : int T

-| val w2 = C("z", size);
val w2 = C Fn  : int T

-| fun f (C(x,g)) = g x;
val f = Fn  : ['a].'a  T -> 'a
```

One nice property of existentially quantified constructors is that their type does not mention the quantified variable. Thus its is possible to construct homogeneous lists of `'b T` objects, even though each one encapsulates a different type `'a`. Note that both `w1` and `w2` both have type `int T` even though `w1` hides the type `int` and `w2` hides the type `string`.

```
-| val L = [w1,w2];
val L = [C (5,fn ),C ("z",fn )] : int T list

-| map f L;
val it = [6,1] : int list
```

When writing a function that pattern matches against a data constructor with existentially quantified components, care must be taken not to use these components in a manner which makes any assumptions about the quantified type. To do so causes a type error:

```
-| fun f (C(x,f)) = x;
Error: Existential argument of Constructor bound in pattern:
C(x,f) is bound in expression: x

-| fun f a (C(x,f)) = (f x,f a);
Error: Existential argument of Constructor bound in pattern:
a is bound in expression: (f x,f a)
```

7.6 Higher Order Kinds

Occasionally one wishes to write a datatype definition which is parameterised over a type constructor, rather than a type. In order to do this one must tell MetaML that the parameter is a type constructor. This is done using *kind* annotations.

For example the second parameter to the datatype `T` below is kinded by the kind `* -> *` indicating that it is a type constructor.

```
-| datatype ( 'a, 'b : ( * -> * ) ) T = C of 'a 'b;
constructor C

-| val q = C[2];
val q = C [2] : (int,list) T
```

Note how the type of `q` includes both a type (`int`) and a type constructor (`list`). Other examples include:

```
-| datatype 'a Tree = Tip of 'a | Node of ('a Tree) * ('a Tree);
constructor Tip
constructor Node

-| val it = C Tip "z" : (string,Tree) T
```

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 = ....`

7.7 Partial Application of Type Constructors

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`

7.8 Monads

A monad is a way of structuring functional programs that makes them easier to modify.

A monad is a type constructor M, and two polymorphic functions $unit : a \to M(a)$ and $bind : M(a) \to (a \to M(b)) \to M(b)$. The way to interpret an expression with type $M(a)$ is as a computation that represents a potential *action* and that also returns a value of type a.

An action might perform I/O, update a mutable variable, or raise an exception. One can implement a monad in a purely functional setting by emulating the actions. This is done by explicitly threading "stores", "I/O streams", or "exception continuations" in and out of all computations. The two polymorphic functions *unit* and *bind* must meet the following three axioms:

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

where $e[x/y]$ is the result of the substitution of the free occurrences of the variable x in e by the variable y.

The monadic operators, *unit* and *bind*, are called the standard morphisms of the monad. The *unit* operator takes a pure value and turns it into an empty action. The *bind* operator sequences two actions. A useful monad will also have non-standard morphisms that describe the primitive actions of the monad (like *read* the value from a variable and *write* a variable in the monad of mutable state).

For more background on the use of monads see [Wad90, Wad95, Wad92].

7.8.1 Monads in MetaML

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

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

This definition uses SML's postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument `('M : * -> * )` of the type constructor `Monad` is itself a unary type constructor [Jon93]. We say that `'M` has *kind*: `* -> *`. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions [OL96]. The type variables in brackets, e.g. `['a,'b]`, are universally quantified . Because of the explicit type annotations in the `datatype` definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the MetaML type inference engine.

In MetaML, `Monad` is a first-class, although *pre-defined* or *built-in* type. In particular, there are two syntactic forms which are aware of the `Monad` datatype: `Do` and `Return`. `Do` and `Return` are MetaML's syntactic interface to the *unit* and *bind* of a monad. We have modeled them after the do-notation of Haskell [HPW92, PH+96]. An important difference is that MetaML's `Do` and `Return` are both parameterized by an expression of type `'M Monad`. `Do` and `Return` are 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 }}}
```

Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. The monad axioms, expressed in METAML's `Do` and `Return` notation are:

```
Do { x <- Return e ; z }          = z[e/x]
Do { x <- m ; Return x }          = m
Do { x <- Do { y <- a ; b } ; c } =
   Do { y' <- a ; Do { x <- b[y'/y] ; c } } =
   Do { y' <- a ; x <- b[y'/y] ; c }
```

Some sample monad definitions follow:

```
-| type 'x Id = 'x;
type Id

-| val Id = (Mon (fn x => x, fn x => fn f => f x)) : Id Monad;
val Id = Mon (Fn ,Fn ) : (FN a => a )  Monad

-| datatype 'a intSt = C of (int -> ('a * int));
constructor C

-| val intSt =
let fun unit x = C(fn n => (x,n))
    fun bind (C x) f = C (fn n => let val (a,n1) = x n
                                      val (C g) = f a
                                  in g n1 end)
in (Mon(unit,bind)) end;
val intSt = Mon (Fn ,Fn ) : intSt Monad

-| datatype 'a option = SOME of 'a | NONE;
constructor SOME
constructor NONE

-| val except =
let fun bind (SOME x) f = f x | bind NONE f = NONE
in (Mon(SOME,bind)) : option Monad end;
val except = Mon (Fn ,Fn ) : option Monad
```

Some sample programs using the `Do` notation in the exception monad are:

```
val q = Do except {a <- Return except 45 ; Return except (a + 2) };
val q = SOME 47 : int option

-| val q = Do except {a <- NONE ; Return except (a + 2) };
val q = NONE  : int option
```

In the integer state monad the result of a monad computation is harder to observe since some thing of type `'a intSt` is a function encapsulated by the `C` data constructor.

```
 -| val q = Do intSt {a <- Return intSt 45 ; Return intSt (a + 2) };
val q = C fn  : int intSt

-| val C f = q;
val f = fn  : int -> (int * int)

-| f 6;
val it = (47,6) : (int * int)
```

8 Experimental Features of MetaML

In this section we discuss several experimental features of MetaML. We include here features we feel are "not quite ready for prime time". Such features may have incomplete implementations, or unsound semantics in the presence of other features, or other problems. We have found most of them useful for something, and include them so that others may benefit or experiment with our "half-baked" ideas.

8.1 Delineated Control Operators

Olivier Danvy and Andreas Filinski describe an mechanism for abstracting control information using a set of operators called `shift` and `reset` [DF90]. In a follow-on paper [Fil94] Filinski shows how these operators can be implemented in Standard ML of New Jersey, using the module system (to encapsulate polymorphism), and the non-standard `callcc` function of the New Jersey implementation.

In MetaML we can express these operators using local polymorphism and a set of built in primitives. A `Control` object is a pair of polymorphic functions `shift` and `reset`.

```
datatype 'ans Control =
  Control of (['a]. (('a -> 'ans) -> 'ans) -> 'a) *
             ((unit -> 'ans) -> 'ans);
constructor Control
```

We encapsulate these using MetaML's using a polymorphic data constructor. Then construct a function which returns a polymorphic instance of one of these control encapsulations from the two MetaML primitives `shift_prim` and `reset_prim`.

```
-| fun SandR () = Control(shift_prim,reset_prim);
val SandR = Fn  : ['a].unit -> 'a  Control
```

Several examples of its use follow:

```
-| let val Control(shift,reset) = SandR ()
in 1 + reset (fn () => 2 * shift (fn k => 3 * ((k 4) + (k 5)))) end;

val it = 55 : int

-| let val Control(shift,reset) = SandR ()
```

```
in 1 + reset (fn () => 2 * shift (fn k => 3 * (k 4))) end;
val it = 25 : int

-| let val Control(shift,reset) = SandR ()
in 1 + reset (fn () => 2 * shift (fn k => k 10)) end;
val it = 21 : int

-| let val Control(shift,reset) = SandR ()
in 1 + reset (fn () => 2 * shift (fn k => 10)) end;
val it = 11 : int

-| let val Control(shift,reset) = SandR ()
in 1 + reset (fn () => 2 * 1) end;
val it = 3 : int

-| let val Control(shift,reset) = SandR ()
in 1 + reset (fn () => 2 * (shift (fn k => 10)) - (shift (fn k1 => k1 3))) end;
val it = 11 : int
```

Warning. The shift and reset operators do not work inside code construction.

8.2 Dynamic type-checking

Warning this section describes an experimental extension to METAML **which may not run as advertised, or may not run at all in the current implementation.**

Type-checking prevents the programmer from writing some incorrect programs. It throws away those programs which apply functions to values for which they are not defined. Mechanisms such as polymorphism allow the system to check and accept more programs. However, when dealing with multistage programs, this type discipline may become too restrictive. We illustrate this point by the function `sumN` below.

```
fun sumN n body =
    if n=0
       then body
       else < fn x => ~(sumN (n-1) < x + ~body >) >;
```

This function is intend to input an integer `n` and compute the code:

```
<fn x1 => fn x2 => ... => fn xn => x1 + x2 + ... + xn>
```

Unfortunately, the type-checker cannot accept `SumN`. This is because its type cannot be inferred at compile-time, it depends on the dynamic value of `n`:

```
-| fun sumN n body =
       if n=0
          then body
          else < fn x => ~(sumN (n-1) < x + ~body >) >;
Type Error:
```

```
  Cannot unify the types: occurs check
   ?
  and
   ? -> ?
 in expression: sumN
```

This becomes crucial when dealing with *reflection*. In order to allow self-applicable interpretation in MetaML this point must be addressed, since the result type of an interpretation depends on the input program, just as the result of `sumN` depends on `n`.

In addition to the normal type checking for code described earlier MetaML offers another mechanism denoted *dynamic type-checking* [SSJ98]. The user may *explicitly* specify that a piece of code is not to be *fully* type-checked statically. This is done by enclosing the frozen expression by *dynamic brackets* `<| |>` instead of usual meta-brackets `< >`. When MetaML's type-checker encounters such an expression, it gives it the type `Dyn`, and it pre-processes it so the actual type can be quickly computed at run-time.

```
-| <| fn x => x |>;
val it = <| fn a => a : 'b -> 'b with () |> : Dyn
```

In the example above, the resulting type is `Dyn`. Note that dynamic code carries it's own type. This is indicated by the `: 'b -> 'b` inside the dynamic code brackets. If this dynamic code is ever used, either in `run`, or spliced using `esc` into a larger piece of code, that a dynamic run-time check will be performed, using this type annotation.

Using dynamic brackets we can write a `sumN` function which MetaML accepts:

```
-| fun sumN n body = if n=0 then body
                             else <| fn x => `(sumN (n-1) (<| (x + `body) |>)) |>;
val sumN = fn : int -> Dyn -> Dyn
```

Back quote (`) denotes dynamic escape. It has to be use instead of escape (~) inside a dynamic bracketed expression. Finally, MetaML also offers a *dynamic run* operator denoted `drun`. `drun` takes to arguments: a dynamic piece of code to be evaluated and an *exception expression*. The exception expression if the dynamic code is not of the appropriate type. Using `drun`, we can execute the function `sumN`:

```
-| (sumN 2 (<|0|>));
val it = <| fn a => fn b => b %+ a %+ 0 : int -> int -> int with () |> : Dyn

-| drun (it, fn x => fn y => ~1);
val it = fn : int -> int -> int

-| it 4 5;
val it = 9 : int
```

In the example above, the function (`fn x => fn y => ~1`) will be used if dynamic type-checking fails. By making the exception function return the particular value `~1` (negative one), we can catch the type error in any call to the dynamic function.

This mechanism, although quite useful, should be used as rarely as possible. There is a strong temptation for the programmer to "fix" some functions by deferring their type-checking. Usually all this accomplishes is that it also defers the errors. Dynamic typing should only be used when functions have types which depend upon the value of their inputs.

It is a goal of SML to check the consistency of expressions completely at compile-time, when possible. Avoiding compile time checking when not really necessary is not a good idea.

8.3 Intensional Analysis of code

There are two important kind of meta-programming scenarios: program generators, and program analyses. A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kind of meta-systems are: program transformers, optimizers, and partial evaluation systems.

Analyses are an important class of meta-programs. Unfortunately there is a conflict between the analysis of code and the efficient construction of code. If one can obeserve the structure of code, then it is possible to distinguish between: `< (fn x => x + 1) 5 >` and `<5 + 1>`, which cannot be distinguished if all we can do is `run` code. We have argued in Section 5 that there are good reasons for a "generator" system to automatically transform the first into the second, thus in a system with analysis of code, it is possible to lose confluence. That is why we include this section on code analysis in the experimental features of MetaML.

If code is just a data structure, like any other, then it should be 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>
```

Pattern matching against code is very useful, and is used to great benefit in the next section.

8.4 A HOL-like Theorem Prover

In this section we describe extensions we made to MetaML in order to build an HOL-loke theorem prover. Our motivation in this endeavor ws three-fold. First, to use MetaML for something other than staging. Second, to stress the implementation, and third to motivate the intensional analysis capabilites described in the previous section. We wanted to know what capabilites wer really necessary, and what capabilites were missing.

Our goals in building the HOL-like system were to hack the current implementation as least as possible, to code everything else using MetaML as the meta-language. This is possible because in MetaML code is a first class object, but will require some intensional analysis. Our last goal was to build a unified system where one can write programs, execute them, and reason about them all in the same system, all using the same language.

Our system is built upon the following ideas. There is an abstract datatype for theorems. The ADT has two kinds of interface functions. *Constructors* - which construct only true theorems. and *combinators* - which construct new theorems from old. The form of a thoerem is: it premises `|-` *conclusion*. The *principle rule* for manipulating theorems is the following: `p |- q => r` can be transformed to `p,q |- r`.

The style of proof supported by the system is to use combinators and constructors to: manipulate the conclusion of a theorem, to produce new theorems, move information into and out of the premises by using the Principle Rule.

When building our system we came to realize some crucial differences between HOL and our system.

- HOL reasons about mathematical sets.
- Our system reasons about ML programs.
- ML Programs may not terminate
- ML functions evaluate their arguments (call be value - language)

We began our implementation by introducing a new `theorem` datatype.

```
datatype theorem =
    Thm of <bool> list * <bool>;

fun conclusion (Thm(p,c)) = c;

fun premises   (Thm(p,c)) = p;
```

Note that we code of type `bool` to represent theorems. The interpretation of a theorem `Thm([p1,p2, ... , pn], conc)` is if `p1, p2, ... , pn` all terminate and evaluate to `true`, then `conc` will terminate and evaluate to `true`.

We can construct several primitives which always return true theorems as follows:

```
fun halts t = true;

fun HALTS_EVAL t =
   if halts(run t)
      then Thm([], <halts ~t>)
      else Thm([],<true>)
(* the second branch of the conditional can't happen *);

fun ASSUME b = Thm ([b],b);
```

The first stage function `halts` evaluates its argument (after all ML is a strict language). It it terminates, it returns true. `HALTS_EVAL`, takes a peice if code, `runs` it and if it terminates, builds a theorem which states this fact. If the piece of code does not terminate, then the theorem is never constructed since in that case `HALTS_EVAL` does not terminate.

The theorem constructor `ASSUME` builds the trivial theorem that if `b` halts then `b` halts.

The theorem combinator `RESOLVE` can be built as follows:

```
fun RESOLVE(t1 as Thm(p,t))(Thm(p',t')) =
 if member t' p
    then Thm(union (delete t' p) p', t)
    else error
      ("Conclusion "^(show t')^
       " is not a member hypoth of\n   "^
       (show t1))
```

Sophisticated constructors and combinators like `RESOLVE` need to analyse their code arguments. In order to do this we needed to provide several code inspecting primitives:

The first primitve was a function for comparing code for equality modulo alpha renaming. We supply this capability via the function `expEq`.

```
-| expEq;
val it = Fn  : ['a].<'a > -> <'a > -> bool

-| expEq <fn a => a + 1> <fn b => b + 1>;
val it = true  : bool

-| expEq <fn a => a + 1> <fn b => 1 + b>;
val it = false  : bool
```

The second was to be able to use the defintions of functions. This is done by the meta-function `expand`. Expand can replace any piece of code which is a lexically captured constant with its definition. Non lexically captured constants are returned unchanged.

```
-| expand;
val it = Fn  : ['a].<'a > -> <'a >

-| expand <length>;
val it =
<(fn [] => 0 | (b::a) => 1 %+ %length a)>
 : <'1  list -> int>

-| expand <3>;
val it = <3> : <int>
```

Note that the expansion of a lexically captured constant me introduce other lexically captured constants.

Many more details of the HOL-like theorem prover can be learned from the power-point presentation *A HOL-like Theorem Prove Using MetaML as a Meta-Language* found on the MetaML home page.

8.5 Reification

Olivier Danvy introduced an interesting way to construct a partial evaluator by using type-directed computation[Dan95]. In 1997 we showed how we could integrate this idea into a meta-programming system with explicit staging annotations[She97]. The crucial operation in type-directed partial evaluation is a type-directed function called *reify*. Given a type, and value with that type, it produces a representation of that value. In our paper we showed how in a language with type inference, the type need not be supplied since, since the inference mechanism can supply it automatically. We have implemented these features in the current version of MetaML. For example:

```
-| fun f x = x + 3;
val f = fn  : int -> int

-| reify f;
val it = <(fn a => a %+ 3)> : <int -> int>
```

The function `reify` is like the annotation `lift`, except that it applies to functions as well as ground types.

Reification can be used to construct lifted versions of functions rather than staging them manually. For example consider writing a staged *append* function whose first argument is static and whose second is dynamic.

```
fun append2 x =
  <fn y => ~(case x of
                [] => <y>
             | w::ws => < ~(lift w) :: ( ~(append2 ws) y) >)>;
-| val append2 = Fn  : ['a^].'a  list -> <'a  list -> 'a  list>

-| append2 [3,4];
val it = <(fn a => 3 :: 4 :: a)> : <int list -> int list>
```

Using reification we can do this simply by applying `reify` to a partially applied call to the existing 1 stage append (`(op @)`) function.

```
-| fn x => [1,2,3] @ x;
val it = fn  : int list -> int list

-| reify it;
val it = <(fn a => 1 :: 2 :: 3 :: a)> : <int list -> int list>
```

While `reify` is useful, it is also dangerous. Reification can easily lead to non-termination, and our implementation of `reify` depends crucially on implementation details which conflict with our implementation of the intensional analysis of code.

Applying reify to any function which uses pattern matching against code (Section 8.3), or uses one of the primitives which observe the structure of code (`force`, `is_code`, `showCode`, etc) can behave in a non-deterministic way.

It is even possible to break the soundness of the type system by mixing these two experimental features. See section 9.18 for details.

9 The MetaMl Library

9.1 Bool functions

Usual boolean values and functions.

```
val false   : bool
val true    : bool
val not     : bool -> bool
val andAlse : (bool * bool) -> bool
val orElse  : (bool * bool) -> bool
```

andAlso and orElse are functions, whereas andalso and orelse are constructs in MetaML used in an infix fashion.

9.2 Num operators and functions

Num type stands for int and float types. It allows overloading the common arithmetic functions, and uses the *num* qualification to restrict the universal polymorphism. A num type variable is denoted by a#.

```
val op +      : ['a#].('a * 'a ) -> 'a
val op -      : ['a#].('a * 'a ) -> 'a
val op *      : ['a#].('a * 'a ) -> 'a
val sign      : ['a#].'a -> int
val min       : ['a#].('a * 'a ) -> 'a
val max       : ['a#].('a * 'a ) -> 'a
val sameSign : ['a#].('a * 'a ) -> bool
val abs       : ['a#].'a -> 'a
```

9.3 Comparison operators

Comparison operators are overloaded. The orderable qualification 'a< contains int, real, char, and string. Contrast this with the equality qualification on type variables (denoted by 'a=) which contains all basic types and datatypes that do not contain function types.

```
val op =      : ['a=].('a  * 'a ) -> bool
val op '>'    : ['a<].('a  * 'a ) -> bool
val op >=     : ['a<].('a  * 'a ) -> bool
val op '<'    : ['a<].('a  * 'a ) -> bool
val op <=     : ['a<].('a  * 'a ) -> bool
val op <>     : ['a=].('a  * 'a ) -> bool
```

9.4 int functions

```
val op div     : (int * int) -> int
val op mod     : (int * int) -> int
val quot       : (int * int) -> int
val rem        : (int * int) -> int
val inteq      : int -> int -> bool
val fromString : string -> int option
```

9.5 real functions

```
val op /        : (real * real) -> real
val sin         : real -> real
val cos         : real -> real
val int2real    : int -> real
val RfromString : string -> real option
val trunc       : real -> int
val round       : real -> int
```

9.6 char functions

```
val ord         : char -> int
val chr         : int -> char
val succ        : char -> char
val pred        : char -> char
val toLower     : char -> char
val toUpper     : char -> char
val isAlpha     : char -> bool
val isAlphaNum  : char -> bool
val isAscii     : char -> bool
val isCntrl     : char -> bool
val isDigit     : char -> bool
val isGraph     : char -> bool
val isHexdigit  : char -> bool
val isLower     : char -> bool
val isPrintr    : char -> bool
val isUpper     : char -> bool
val isSpace     : char -> bool
val isPunct     : char -> bool
val CtoString   : char -> string
val CfromString : string -> char option
```

9.7 string functions

```
val size        : string -> int
val sub         : (string * int) -> char
val substring   : string -> int -> int -> string
val op ^        : (string * string) -> string
val str         : char -> string
val streq       : string -> string -> bool
val cat         : string -> string -> string
val Sconcat     : string list -> string
val implode     : char list -> string
```

```
val explode     : string -> char list
val isPrefix    : string -> string -> bool
val contains    : string -> char -> bool
val notContains : string -> char -> bool
val toString : ['a<]. 'a -> string (* works on int, string, char and real *)
```

9.8 List functions

```
val nil       : ['a].'a  list
val op ::     : ['a].('a  * 'a  list) -> 'a  list
val tl        : ['a].'a  list -> 'a  list
val hd        : ['a].'a  list -> 'a
val null      : ['a].'a  list -> bool
val op @      : ['a].('a  list * 'a  list) -> 'a  list
val append    : ['a].'a  list -> 'a  list -> 'a  list
val length    : ['a].'a  list -> int
val rev       : ['a].'a  list -> 'a  list
val revappend : ['a].('a  list * 'a  list) -> 'a  list
val drop      : ['a].('a  list * int) -> 'a  list
val take      : ['a].('a  list * int) -> 'a  list
val concat    : ['a].'a  list list -> 'a  list
val nth       : ['a].('a  list * int) -> 'a
val last      : ['a].'a  list -> 'a
val map       : ['a,'b].('b  -> 'a ) -> 'b  list -> 'a  list
```

9.9 Vector primitives

```
val Vget      : ['a].('a  vector * int) -> 'a
val Vsub      : ['a].('a  vector * int) -> 'a
val Vlength   : ['a].'a  vector -> int
val VfromList : ['a].'a  list -> 'a  vector
val Vconcat   : ['a].'a  vector list -> 'a  vector
val Vextract  : ['a].'a  vector -> int -> int option -> 'a  vector
```

9.10 Ref primitives

```
val ref   : ['a].'a  -> 'a  ref
val ! : ['a].'a  ref -> 'a
val op := : ['a].('a  ref * 'a ) -> unit
val unref : ['a].'a  ref -> 'a    (* like !, included for compatibility *)
                                  (* with previous versions of MetaML   *)
```

9.11 option type

```
datatype 'a option = SOME of 'a | NONE
```

```
val SOME : ['a]. 'a -> 'a option
val NONE : ['a]. 'a option
```

9.12 sum type

```
datatype ('a,'b) sum = Right of 'a | Left of 'b
val Right : ['a,'b]. 'b -> ('b,'a) sum
val Lift : ['a,'b]. 'b -> ('a,'b) sum
```

9.13 monad type

```
datatype ('T: * -> * ) Monad =
     Mon of (['a]. 'a -> 'a 'T) * (['a,'b]. 'a 'T -> ('a -> 'b 'T) -> 'b 'T);
```

9.14 I/O

```
val stdIn         : instream
val stdOut         : outstream
val stdErr         : outstream
val openIn        : string -> instream
val openOut       : string -> outstream
val openAppend    : string -> outstream
val openString    : string -> instream
val closeOut      : outstream -> unit
val closeIn       : instream -> unit
val print         : string -> unit
val inputLine     : instream -> string
val outputString : (outstream * string) -> unit
val flushOut      : outstream -> unit
val endOfStream   : instream -> bool
val cd            : string -> unit
val pwd           : unit -> string
val ls            : unit -> unit
val dir           : unit -> unit    (* synonym of ls function *)
val exit          : unit -> unit
```

9.15 MetaML System interfaces

```
val printEnv      : unit -> unit       (* describe predefined functions *)
val use           : string -> unit
val show_type   : bool ref             (* to print types in more detail *)
val pp          : ['a].'a  -> unit     (* Attempts to print anything    *)
val feature : int -> bool
```

The function `feature` controls optimizations that are performed on code. Type `feature 0` to read the status of whether these optimizations are performed or not, and also to read the associated integer tag that switch them on and off.

9.16 General functions

```
val op o       : ['a,'b,'c].(('c  -> 'b ) * ('a  -> 'c )) -> 'a  -> 'b
val K          : ['a,'b].'b  -> 'a  -> 'b
val id         : ['a].'a  -> 'a
val error      : ['a].string -> 'a
val Bottom     : exn
val bot        : ['a].'a
val show       : ['a].'a  -> string
```

9.17 Code primitives

```
val lift      : ['a^].'a  -> <'a >
val expand    : ['a].<'a > -> <'a >
val sc        : ['a].<'a > -> string
val showCode  : ['a].<'a > -> string       (* same function as sc *)
```

The function `lift` applies to types that are liftable. The liftable types correspond to the equality types. `expand` attempts to inline cross-stage persistence values, When it is not possible it returns the same code. See section 8.4.

9.18 Unsafe functions

The first two functions below correspond to *shift* and *reset* control operators, see section 8.1. They have an unsafe interaction with code, an unexpectable behavior might occur when they are combined. The function `cast` is obviously unsound with respect to the type system, its use will probably produce a run time error.

```
val shift_prim : ['a,'b].(('a  -> 'b ) -> 'b ) -> 'a
val reset_prim : ['a,'b].(unit -> 'b ) -> 'b
val cast        : ['a,'b].'a  -> 'b
val force      : ['a,'b].<'b  -> 'a > -> <'b > -> <'a >
val is_code    : ['a].'a  -> bool
val is_lambda : ['a,'b].<'a  -> 'b > -> bool
```

`force` is a built-in function, it has different behavior than the following function which just builds an application.

```
-| fun beta f x =  <~f ~x>
val beta = Fn  : ['a,'b].<'b  -> 'a > -> <'b > -> <'a >
```

The following example demonstrates this difference:

```
-| force  <fn x => x + x > <3+4> ;
 val it = <3 %+ 4 %+ 3 %+ 4> : <int>

-| beta <fn x => x + x > <3+4> ;
val it =  <((fn a => a %+ a)) (3 %+ 4)> : <int>

-| let val f = <fn x => x + x >
       val x = <3+4>
   in <~f ~x> end;
val it = <((fn a => a %+ a)) (3 %+ 4)> : <int>

-| let val f = <fn x => x + x >
       val x = <3+4>
   in force f x end;
val it = <3 %+ 4 %+ 3 %+ 4> : <int>
```

Obviously force is not sound with *effects* since it forces the application of its first argument causing duplication of the its second argument. This mean that it does not preserve the semantics when effects are used. The safe-beta optimization, see Section 5.1 does preserve semantics.

The reify function lifts programs at level 0 to code (level 1)

```
val reify     : ['a]. 'a -> <a>
```

It is not safe to reify functions that manipulate code or do intentional analysis on code. Unexpected behavior might happen, for instance:

```
-| reify ( fn x => is_code x)
val it =
<(fn a => true)>  : <'1  -> bool>
-| reify (fn x => show x)
val it =
<(fn a => "<x'566>")>  : <'1  -> string>
-| reify (fn x => sc x);
val it =
<(fn a => "x'570")> : <<'1 > -> string>
```

Mixing reify and intensional analyis of code might corrupt the type system. The following examples demonstrate this.

```
-| reify (fn x => fn y => force x y );
val it = <(fn a => a)> : <<'1  -> '2 > -> <'1 > -> <'2 >>

-| reify (fn x => reify x);
val it =  <(fn a => a)> : <'1  -> <'1 >>
```

9.19 Higher-Order Logic (HOL) theorem prover functions

```
val haltsPrim : ['a].<'a > -> bool
val expEq     : ['a].<'a > -> <'a > -> bool
val quant1    : ['a,'b].(<'a > -> 'b ) -> 'b
val quant2    : ['a,'b,'c].(<'a > -> <'b > -> 'c ) -> 'c
val quant3    : ['a,'b,'c,'d].(<'a > -> <'b > -> <'d > -> 'c ) -> 'c
val verify1 : ['a].(<'a > -> theorem) -> <'a > -> theorem
val verify2 : ['a,'b].(<'a > -> <'b > -> theorem) -> <'a > -> <'b > -> theorem
val Thm     : (<bool> list * <bool>) -> theorem
val box     : ['a].string -> 'a  -> 'a
val update    : ['a,'b].<'a > -> int list -> <'b > -> <'a >
val get       : ['a,'b].<'a > -> int list -> <'b >
val search    : ['a,'b].<'a > -> <'b > -> int list list
```

References

[Dan95] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the ACM Symposium on Principle of Programming Languages*, pages 242–257. ACM, 1995.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[Fil94] Andrzej Filinski. Representing monads. In *the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, Jan 1994.

[HPW92] Paul Hudak, S. L. Peyton Jones, and Philip Wadler. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[Jon93] 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.

[Ler97] Xavier Leroy. *The Caml Light system, documentation and user's guide, realize 0.73.* http://pauillac.inria.fr/caml/man-caml/index.html, 1997.

[MTBS98] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, Oregon Graduate Institute, October 1998. `ftp://cse.ogi.edu/pub/tech-reports/`.

[MTBS99] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). In *European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*. Springer-Verlag, 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT-Press, 1997.

[NN93] Flemming Nielson and Hanne Riis Nielson. number 34 in cambridge tracts in theoretical computer science. In *Two-level functional languages*. Cambridge University Press, 1993.

[OL96] 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.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer (2nd Edition).* Cambridge University Press, 1996.

[PH+96] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value- and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[She97] Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, page ?? ACM, 1997.

[SSJ98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, jan 1998.

[SW97] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 9(6):916–941, Nov 1997.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Application.* PhD thesis, Oregon Graduate Institute, November 1999.

[TBS98] W. Taha, Z. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th ICALP*, Aalborg, Denmark, 1998.

[TS97] 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*, page ?? ACM, 1997.

[Wad90] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Program ming, Nice, France*, pages 61–78, June 1990.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. (This is a revised version of [**?**].).

[Wri95] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.