MetaML and Multi-Stage Programming with Explicit Annotations*

Walid Taha and Tim Sheard

Oregon Graduate Institute

Abstract. We introduce MetaML, a practically-motivated, statically-typed multi-stage programming language. MetaML is a "real" language. We have built an implementation and used it to solve multi-stage problems.

MetaML allows the programmer to construct, combine, and execute code fragments in a type-safe manner. Code fragments can contain free variables, but they obey the static-scoping principle. MetaML performs type-checking for all stages once and for all before the execution of the first stage.

Certain anomalies with our first MetaML implementation led us to formalize an illustrative subset of the MetaML implementation. We present both a big-step semantics and type system for this subset, and prove the type system's soundness with respect to a big-step semantics. From a software engineering point of view, this means that generators written in the MetaML subset never generate unsafe programs. A type system and semantics for full MetaML is still ongoing work.

We argue that multi-stage languages are useful as programming languages in their own right, that they supply a sound basis for high-level program generation technology, and that they should support features that make it possible for programmers to write staged computations without significantly changing their normal programming style. To illustrate this we provide a simple three stage example elaborating a number of practical issues.

The design of MetaML was based on two main principles that we identified as fundamental for high-level program generation, namely, cross-stage persistence and cross-stage safety. We present these principles, explain the technical problems they give rise to, and how we address with these problems in our implementation.

Keywords: Functional Programming, λ-Calculus, High-level Program Generation, Type-Safety, Type-Systems, Programming Language Semantics, Multi-level Languages, Multi-stage Languages.

* The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-93-C-0069, and NSF Grant IRI-9625462. An earlier version of this paper appeared in The Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation. pp 203-217. Amsterdam, The Netherlands, June 12-13, 1997.

1 Introduction

High-level program generators can increase the efficiency, productivity, reliability, and quality of software systems [26, 22, 23]. Despite the numerous examples of program generation in the literature, almost all these systems construct program fragments using *ad hoc* techniques.

Our thesis is that a well-designed statically-typed multi-stage programming language supplies a sound basis for high-level program generation technology. Our goal is to design a language that allows the user to construct, combine, and evaluate programs at a higher level of abstraction than the classic "programs-as-strings" level. Using such a language would make the formal verification of generated-program properties easier.

1.1 Staging and Multi-Stage Programming

The concept of a stage arises naturally in a wide variety of situations. For a compiled language, the execution of a program involves two distinct stages: compile-time, and run-time. Three distinct stages appear in the context of program generation: generation, compilation, and execution. For example, the Yacc parser generator first reads a grammar and generates C code; second, this program is compiled; third, the user runs the compiled code.

A multi-stage program is one that involves the generation, compilation, and execution of code, all inside the same process. Multi-stage languages express multi-stage programs. Staging, and consequently multi-stage programming, address the need for general purpose solutions which do not pay run-time interpretive overheads. Many studies have demonstrated the effectiveness of staging [2, 17, 16, 8, 12, 25, 37, 49]. Yet there has generally been little support for *writing* multi-stage programs directly in high level programming languages such as SML or Haskell. Recently, multi-level languages have been proposed as intermediate an representation for partial evaluation [13, 9, 10], and as a formal foundation for run-time code generation [7]. In this paper we hope to show that a carefully designed multi-level language would also be well-suited for multi-stage programming.

1.2 MetaML

MetaML is an SML-like language with special constructs for multi-stage programming. MetaML is *tightly integrated* in that programs can be constructed, combined, compiled, and executed all under a single paradigm. Programs are represented as abstract syntax trees in a manner that avoids going through string representations. This makes verifying semantic properties of multi-stage programs possible. The key features of MetaML are as follows:

- Staging annotations: Four distinct constructs which we believe are a good basis for general-purpose multi-stage programming.

- Static type-checking: A multi-stage program is type-checked once and for all before it begins executing, ensuring the safety of all computations in all stages.
- Cross-stage persistence: A variable bound in a particular stage, will be available in futures stages.
- Cross-stage safety: An input first available in a particular stage cannot be used at an earlier stage.
- Static scoping of variables in code fragments.

2 Relationship to LISP

MetaML has three annotations, Brackets, Escape, and Run, that are analogous to LISP's back-quote, comma, and eval constructs. This analogy is useful if the reader is familiar with LISP. Brackets are similar to back-quote. Escape is similar to comma. Run is similar to eval in the empty environment. However, the analogy is not perfect. LISP does not ensure that variables (atoms) occurring in a back-quoted expressions are bound according to the rules of static scoping. For example '(plus 3 5) does not bind plus in the scope where the term occurs. We view this as an important feature of MetaML. We view MetaML's semantics as a concise formalization of the semantics of LISP's three constructs, but with static scoping. This is similar in spirit to Brian Smith's semantically motivated LISP [45, 46]. Finally, whereas LISP is dynamically typed, MetaML is statically typed.

The annotations can also be viewed as providing a simple but statically-typed macro-expansion system. This will become clear as we introduce and demonstrate the use of these constructs. But it is also important to note that the annotations don't allow the definition of new language constructs or binding mechanisms, as is sometimes expected from macro-expansion systems.

Finally, we should point out that back-quote and comma are macros in LISP. This leads to two problems. First, they have non-trivial formal semantics (about two pages of LISP code). Second, because of the way they expand at parse-time, they can lead to a representation overhead exponential in the number of levels in a multi-level program [10]. MetaML avoids both problems by a direct treatment of Bracket and Escape as language constructs.

3 Relationship to Linguistic Reflection

"Linguistic reflection is defined as the ability of a program to generate new program fragments and to integrate these into its own execution" [47]. MetaML is a descendent of CRML [40, 41, 15], which in turn was greatly influenced by TRPL [38, 39]. All three of these languages support linguistic reflection. Both CRML and TRPL were two-stage languages that allowed users to provide compile-time functions (much like macros) which directed the compiler to perform compile-time reductions. Both emphasized the use of computations over representations

of a program's datatype definitions. By generating functions from datatype definitions, it is possible to create specific instances of generic functions like equality functions, pretty printers, and parsers [39]. This provides an abstraction mechanism not available in traditional languages. MetaML improves upon these languages by adding hygienic variables, generalizing the number of stages, and emphasizing the soundness of its type system.

4 Relationship to Partial Evaluation

Today, the most sophisticated automatic staging techniques are found in partial evaluation systems [20]. Partial evaluation optimizes a program using partial information about the program's inputs. The goal is to identify and perform as many computations as possible before run-time.

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

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

The relationship between partial-evaluation and multi-stage programming is that the intermediate data structure between the two steps is a *two-stage annotated program* [1, 34], and that the specialization phase is the first stage in the execution of the two-stage annotated program produced by BTA. Recently, Glück and Jørgensen proposed *multi-level BTA* and showed that it is an efficient alternative to multiple specialization [9, 10]. Their underlying annotated language is closely related to MetaML, but without static-typing.

5 Why Explicit Annotations?

While BTA performs staging automatically, there are a number of reasons why the manual staging of programs is both interesting and desirable:

Pragmatic. The subtlety of the semantics of annotated programs warrants studying them in relative isolation, and without the added complexity of other partial evaluation issues such as BTA.

As a Pedagogical Tool. It has been observed that it is sometimes hard for users to understand the workings of partial evaluation systems [18]. New users often lack a good mental model of how partial evaluation systems work. Furthermore, new users are often uncertain: What is the output of a binding-time analysis? What are the annotations? How are they expressed? What do they really mean? The answers to these questions are crucial to the effective use of partial evaluation. Although BTA is an involved process, requiring special expertise, the annotations it produces are relatively simple and easy to understand.

Our observation is that programmers can understand the annotated output of BTA, without actually knowing how BTA works. Having a programming language with explicit staging annotations would help users of partial evaluation understand more of the issues involved in staged computation, and, hopefully, reduce the steep learning curve currently associated with learning to use a partial evaluator effectively [20].

For Controlling Evaluation Order. Whenever performance is an issue, control of evaluation order is important. BTA optimizes the evaluation order given the time of arrival of inputs, but sometimes it is just easier to say what is wanted, rather than to force a BTA to discover it [19]. Automatic analyses like BTA are necessarily incomplete, and can only approximate the knowledge of the programmer. By using explicit annotations the programmer can exploit his full knowledge of the program domain. In a language with automatic staging, having explicit annotations can offer the programmer a well designed back-door for dealing with instances when the analysis reaches its limits.

For Controlling Termination Behavior. Annotations can alter termination behavior in two ways: 1) Specialization of an annotated program can fail to terminate, and 2) the generated program itself might have termination behavior differing from that of the original program [20]. While this is an area of active investigation in partial evaluation, programming with explicit annotation gives the user complete control over (and responsibility for) termination behaviour in a staged system.

6 MetaML's Staging Annotations

MetaML has four staging annotations: Brackets $\langle\_\rangle$, Escape ~_, Run run _, and Lift lift _. An expression $\langle e \rangle$ builds a piece of code which is a representation of e. An expression ~e splices the code obtained by evaluating e into the body of a surrounding Bracketed expression. An expression ~e is only legal within lexically enclosing Brackets. An expression run e evaluates e to obtain a piece of code, and then evaluates that code. The expression lift e evaluates e to a value v, and then constructs a piece of code representing v. The term e must have ground type. A *ground* type is a type not containing a function type. To illustrate, consider the script of a small MetaML session below[1]:

```
-| val triple = (3+4, ⟨3+4⟩, lift 3+4);
val triple = (7,⟨3 %+ 4⟩,⟨7⟩) : (int * ⟨int⟩ * ⟨int⟩)

-| fun f (x,y,z) = ⟨8 - ~y⟩;
val f = fn : ('a * ⟨int⟩ * 'b) → ⟨int⟩

-| val code = f triple;
val code = ⟨8 %- (3 %+ 4)⟩ : ⟨int⟩
```

[1] The reader should treat the percentage signs %_ as white space until they are explained in the next section.

```
-| run code;
val it = 1 : int
```

The first declaration defines a variable triple. The addition in the first component of the triple is evaluated. The evaluation of the addition in the second component is deferred by the Brackets. The addition in the third component is evaluated and then the result is Lifted into a piece of code. Brackets in types such as ⟨int⟩ are read *"Code of int"*, and distinguish values such as ⟨3+4⟩ from values such as 7. The second declaration illustrates that code can be abstracted over, and that it can be spliced into a larger piece of code. The third declaration applies the function f to triple performing the actual splicing. And the last declaration evaluates this deferred piece of code.

MetaML can be used to construct larger pieces of code at run-time:

```
-| fun mult x n = if n=0
                  then ⟨1⟩
                  else ⟨~x * ~(mult x (n-1))⟩;
val mult = fn : ⟨int⟩ → int → ⟨int⟩

-| val cube = ⟨fn y ⇒ ~(mult ⟨y⟩ 3)⟩;
val cube = ⟨fn a ⇒ a %* (a %* (a %* 1))⟩ : ⟨int → int⟩

-| fun exponent n = ⟨fn y ⇒ ~(mult ⟨y⟩ n)⟩;
val exponent = fn : int → ⟨int → int⟩
```

The function mult, given an integer piece of code x and an integer n, produces a piece of code that is an n-way product of x. This can be used to construct the code of a function that performs the cube operation, or generalized to a generator for producing an exponentiation function from a given exponent n. Note how the looping overhead has been removed from the generated code.

6.1 Roles

Both Lift and Brackets create pieces of code. The essential difference is that Lift evaluates its argument, and Bracket does not. Function values cannot be lifted into code using Lift, as we cannot derive a high-level intensional representation for them in general. However, as we will see, function values can be injected into code fragments using Brackets.

Escape allows us to "evaluate under lambda". This can be seen in the definition of the functions cube and exponent above.

Having Run in the language implies introducing a kind of reflection [45, 3], and allows a delayed computation to be activated.

6.2 Syntactic Precedence Issues

The Escape operator has the highest precedence; even higher than function application. This allows us to write: ⟨f ~x y⟩ rather than ⟨f (~x) y⟩. The Lift and

Run operators have the lowest precedence. The scope of these operators extends to the right as far as possible. This makes it possible to write ⟨f ~(lift g y) z⟩ rather than ⟨f ~(lift (g y)) z⟩.

7 The Design of MetaML

MetaML was designed as a statically-typed programming language, and not as an internal representation for a multi-stage system. Our primary goals for MetaML were: first, it should be suitable for writing multi-staged programs, second it should be as flexible as possible, and third it should ensure that only "reasonable things" can be done using the annotations. Therefore, our design choices were different from those of other multi-stage systems.

To define the semantics of MetaML, a syntactic notion of level is needed. The *level* of an expression is the number of surrounding Brackets, minus the number of surrounding Escapes. It is possible to use a variable at a level different than the level of the lambda-abstraction which binds it. In this sections, we discuss two principles for determining which uses are acceptable, and which are not.

7.1 Cross-Stage Persistence

Cross-stage persistence is one of the distinguishing feature of MetaML. To our knowledge, it has not been proposed or incorporated into any multi-stage programming language. In essence, cross-stage persistence allows the programmer to use a variable bound at the current level in any expression to be executed in a future stage. We believe this to be a desirable and natural property in a multi-stage language. The type system will have to ensure that these variables are available before this expression is evaluated.

When the level of the use of a variable is greater than the level at which it was bound, we say that variable is *cross-stage persistent*.

To the user, cross-stage persistence means the ability to stage expressions that use variables defined at a previous stage. Bracketed expressions with free variables, like lambda-abstractions with free variables, must resolve their free variable occurrences in the static environment where the expression occurs. One can think of a piece of code as containing an environment which binds its cross-stage persistent variables. For example the program

$$\mathsf{val\ a} = 1{+}4\ ;\ \langle 72{+}\mathsf{a}\rangle$$

computes the code fragment ⟨72 %+ %a⟩. The percentage sign %_ indicates that the cross-stage persistent variables a and + are bound in the code's local environment. The variable a has been bound during the first stage to the constant 5. The percentage sign is printed by the display mechanism to indicate that %a is not a variable, but rather, a new *constant*. The name "a" is only provided as a hint to the user about where this new constant originated from. When %a is evaluated in a later stage, it will return 5 independently of the binding for

the variable a in the new context since it is bound in the value's local environment. Arbitrary values (including functions) can be injected into a piece of code using this hygienic binding mechanism. Formally specifying this behavior in a big-step semantics turns out to be non-trivial. In an interpreter for a multi-stage language, this behaviour manifests itself as complex variable-binding rules, the use of closures, or capture-free substitutions. Our implementation semantics addresses cross-stage persistence in a novel way (Section 13.1).

Cross-Platform Portability For high-level program generation, cross-stage persistence comes at a price. Because most compilers do not maintain a high-level representation for values at run-time, being able to inject any value into the code type means that some parts of this code fragment may not be printable. So, if the first stage is performed on one computer, and the second on another, we must "port" the local environment from the first machine to the second. Since arbitrary objects, such as functions and closures, can be bound in this local environment, this can cause portability problems. Currently, MetaML assumes that the computing environment does not change between stages. This is part of what we mean by having an integrated system. Thus, MetaML currently lacks *cross-platform portability*. The loss of this property is the price paid for cross-stage persistence.

Cross-platform portability is usually not an issue for run-time code generation systems, and hence, cross-stage persistence might in fact be more appropriate for such systems. On the other hand, the problem of cross-platform portability is similar to that of lifting functional values in partial evaluation, and type-directed partial evaluation may provide a solution to this problem [4, 42].

7.2 Cross-Stage Safety

Not every staged form of a typable expression should be typable in a multi-stage language. When a variable is used at a level less than the level of the lambda-abstraction in which it is bound, we say the use violates *cross-stage safety*. Cross-stage safety prevents us from staging programs in unreasonable ways, as is the case in the expression

$$\mathsf{fn\ a} \Rightarrow \langle \mathsf{fn\ b} \Rightarrow \tilde{}(\mathsf{a+b}) \rangle$$

Operationally, these annotations dictate computing a+b in the first stage, while the value of b will be available only in the second stage! Therefore, MetaML's type system was designed to ensure that "well-typed programs won't go wrong", where going wrong now includes the violation of the cross-stage safety condition, as well as the standard notions of going wrong [27] in statically-typed languages.

In our experience with MetaML, having a type system to screen out programs containing this kind of error is a significant aid in hand-staging programs.

8 Hand-Staging: A Short Example

Using MetaML, the programer can stage programs by inserting the proper annotations at the right places in the program. The programmer uses these annotations to modify the default (strict) evaluation order of the program.

In our experience, starting with the type of the function to be hand-staged makes the number of different ways in which it can be annotated quite tractable. This leads us to believe that the location of the annotations in a staged version of a program is significantly constrained by its type. For example, consider the function member defined as follows[2]:

```
(* member : int → int list → bool *)
fun member v l =
    if (null l)
      then false
      else if v=(hd l)
           then true
           else member v (tl l);
```

The function member has type int → int list → bool. A good strategy for hand annotating a program is to first determine the target type of the desired annotated program. Suppose the list parameter l is available in the first stage, and the element searched for will be available later. One target type for the hand-staged function is ⟨int⟩ → int list → ⟨bool⟩.

Now we can begin annotating, starting with the whole expression, and working inwards until all sub-expressions are covered. At each step, we try to find the annotations that will "fix" the type of the expression so that the whole function has a type closer to the target type. The following function realizes this type:

```
(* member : ⟨int⟩ → int list → ⟨bool⟩ *)
fun member v l =
    if (null l)
      then ⟨false⟩
      else ⟨if ~v=~(lift hd l)
              then true
              else ~(member v (tl l))⟩;
```

But not all annotations are explicitly dictated by the type. The annotation ~(lift hd l) has the same type as (and replaces) hd l in order to ensure that hd is performed during the first stage. Otherwise, all selections of the head element of the list would have been delayed until the code constructed was Run in a later stage.

The Brackets around the branches of the outermost if-expression ensure that the return value of member will be a code type ⟨_⟩. The first branch ⟨false⟩ needs no further annotations, and makes the return value precisely a ⟨bool⟩. Moving

[2] Function "=" has type (int * int) → bool which forces v and l to have types int and int list, respectively.

inwards in the else branch, the condition of the inner if-expression (in particular ~v) forces the type of the v parameter to have type ⟨int⟩ as planned.

Just like the first branch of the outer if-statement, the inner if-statement must return bool. So, the first branch (true) is fine. But because the recursive call to member has type ⟨bool⟩, it must be Escaped. Inserting this Escape also implies that the recursion will be performed in the first stage, which is exactly the desired behavior. Thus, the result of the staged member function is a recursively-constructed piece of code with type bool.

Evaluating ⟨fn x ⇒ ~(member ⟨x⟩ [1,2,3])⟩ yields:

```
⟨fn d1 ⇒
if d1 %= 1
  then true
  else if d1 %= 2
         then true
         else if d1 %= 3
                then true
                else false⟩
```

9 Back and Forth: Two Useful Functions on Code Types

While staging programs, we found an interesting pair of functions to be useful:

```
(* back: ⟨'a⟩ → ⟨'b⟩ → ⟨'a → 'b⟩ *)
fun back f = ⟨fn x ⇒ ~(f ⟨x⟩)⟩;
(* forth: ⟨'a → 'b⟩ → (⟨'a⟩ → ⟨'b⟩) *)
fun forth f x = ⟨~f ~x⟩;
```

We used a similar construction to stage the member function of type ⟨int⟩ → int list → ⟨bool⟩, within the term ⟨fn x ⇒ ~(member ⟨x⟩ [1,2,3])⟩ which has type ⟨int → bool⟩.

In our experience annotating a function to have type ⟨'a⟩ → ⟨'b⟩ requires less annotations than annotating it to have type ⟨'a → 'b⟩ and is often easier to think about. Because we are more used to reasoning about functions, this leads us to avoid creating functions of the latter kind except when we need to see the code. This also applies to programs with more than two stages. Consider the function:

```
(* back2 : (⟨'a⟩ → ⟨⟨'b⟩⟩ → ⟨⟨'c⟩⟩) → ⟨'a → ⟨'b → 'c⟩⟩ *)
fun back2 f = ⟨fn x ⇒ ⟨fn y ⇒ ~~(f ⟨x⟩ ⟨⟨y⟩⟩)⟩⟩;
```

This allows us to write a program which takes a ⟨a⟩ and a ⟨⟨b⟩⟩ as arguments and which produces a ⟨⟨c⟩⟩, and stage it into a three-stage function. Our experience is that such functions have considerably fewer annotations, and are easier to think about. This is illustrated in the following section.

Another reason for our interest in back and forth is that they are similar to two-level η-expansion [5]. In MetaML, however, back and forth are not only meta-level concepts or optimizations, but rather, first class functions in the language, and the user can apply them directly to values of the appropriate type.

We also conjecture that back and forth form an isomorphism between two interesting subsets of the types ⟨'a⟩ → ⟨'b⟩ and ⟨'a → 'b⟩. These subsets must exclude, for example, non-terminating functions in the set for ⟨'a → 'b⟩. We hope to be able to confirms this conjecture in future work.

10 A Multi-Stage Example

When information arrives in multiple phases it is possible to take advantage of this fact to get better performance. Consider a generic function for computing the inner product of two vectors. In the first stage the arrival of the size of the vectors offers an opportunity to specialize the inner product function on that size, removing the overhead of looping over the body of the computation n times. The arrival of the first vector affords a second opportunity for specialization. If the inner product of that vector is to be taken many times with other vectors it can be specialized by removing the overhead of looking up the elements of the first vector each time. This is exactly the case when computing the multiplication of 2 matrixes. For each row in the first matrix, the dot product of that row will be taken with each column of the second. This example has appeared in several other works [9, 24]. We give three versions of the inner product function. One (iprod) with no staging annotations, the second (iprod2) with two levels of annotations, and the third (iprod3) with two levels of annotations but constructed with the back2 function. In MetaML we quote relational operators involving less-than $<$ and greater-than $>$ because of the possible confusion with Brackets.

```
(* iprod : int → Vector → Vector → int *)
fun iprod n v w =
    if n '>' 0
      then ((nth v n) * (nth w n)) + (iprod (n-1) v w)
      else 0;

(* iprod2 : int → ⟨Vector → ⟨Vector → int⟩⟩ *)
fun iprod2 n = ⟨fn v ⇒ ⟨fn w ⇒
    ~~(if n '>' 0
         then ⟨⟨ (~(lift nth v n) * (nth w n)) + (~(~(iprod2 (n-1)) v) w)⟩⟩
         else ⟨⟨0⟩⟩)⟩⟩;

(* p3 : int → ⟨Vector⟩ → ⟨⟨Vector⟩⟩ → ⟨⟨int⟩⟩ *)
fun p3 n v w =
    if n '>' 0
      then ⟨⟨ (~(lift nth ~v n) * (nth ~~w n)) + ~~(p3 (n-1) v w)⟩⟩
      else ⟨⟨0⟩⟩;

fun iprod3 n = back2 (p3 n);
```

Notice that the staged versions are remarkably similar to the unstaged version, and that the version written with back2 has fewer annotations. The type infer-

ence mechanism and the interactive environment were a great help in placing the annotations correctly.

An important feature of MetaML is the visualization help that the system affords. By testing iprod2 on some inputs we can immediately see the results.

```
val f1 = iprod3 3;
f1 : ⟨Vector → ⟨Vector → int⟩⟩ =
⟨fn d1 ⇒
    ⟨fn d5 ⇒
        (~(lift %nth d1 3) %* (%nth d5 3)) %+
        (~(lift %nth d1 2) %* (%nth d5 2)) %+
        (~(lift %nth d1 1) %* (%nth d5 1)) %+
        0⟩⟩
```

When this piece of code is Run it will return a function, which when applied to a vector builds another piece of code. This building process includes looking up each element in the first vector and splicing in the actual value using the Lift operator. Using Lift is especially valuable if we wish to inspect the result of the next phase. To do that we evaluate the code by Running it, and apply the result to a vector.

```
val f2 = (run f1) [1,0,4];
f2: ⟨Vector → int⟩ =
⟨fn d1 ⇒ (4 %* (%nth d1 3)) %+
          (0 %* (%nth d1 2)) %+
          (1 %* (%nth d1 1)) %+ 0⟩
```

Note how the actual values of the first array appear in the code, and how the access function nth appears as a constant expression applied to the second vector d1.

While this code is good, it does not take full advantage of all the information known in the second stage. In particular, note that we generate code for the third stage which may contain multiplication by 0 or 1. These multiplications can be optimized. To do this we write a second stage function add which given an index into a vector i, an actual value from the first vector x, and a piece of code which names the second vector y, constructs a piece of code which adds the result of the x and y multiplication to the code valued fourth argument e. When x is 0 or 1 special cases are possible.

```
(* add : int → int → ⟨Vector⟩ → ⟨int⟩ *)
fun add i x y e =
    if x=0
      then e
      else if x=1
            then ⟨(nth ~y ~(lift i)) + ~e⟩
            else ⟨(~(lift x) * (nth ~y ~(lift i))) + ~e⟩;
```

This specialized function is now used to build the second stage computation:

```
(* p3 : int → ⟨Vector⟩ → ⟨⟨Vector⟩⟩ → ⟨⟨int⟩⟩ *)
fun p3 n v w =
if n = 1
  then ⟨⟨~(add n (nth ~v n) ~w ⟨0⟩)⟩⟩
  else ⟨⟨~(add n (nth ~v n) ~w ⟨~~(p3 (n-1) v w)⟩)⟩⟩;

fun iprod3 n = back2 (p3 n);
```

Now let us observe the result of the first stage computation.

```
val f3 = iprod3 3;
f3: ⟨Vector → ⟨Vector → int⟩⟩ =
⟨fn d1 ⇒
    ⟨fn d5 ⇒
        ~(%add 3 (%nth d1 3) ⟨d5⟩
                ⟨ ~(%add 2 (%nth d1 2) ⟨d5⟩
                        ⟨~(%add 1 (%nth d1 1) ⟨d5⟩
                                ⟨0⟩)⟩)⟩)⟩⟩
```

This code is linear in the size of the vector; if we had actually in-lined the calls to add it would be exponential. This is another reason why having cross-stage persistent constants (such as add) in code is indispensable. Now let us observe the result of the second stage computation:

```
val f4 = (run f3) [1,0,4];
f4: ⟨Vector → int⟩ = ⟨fn d1 ⇒ (4 %* (%nth d1 3)) %+ (%nth d1 1) %+ 0⟩
```

Note that now only the multiplications that contribute to the answer remain in the third stage program. If the vector is sparse then this sort of optimization can have dramatic effects.

11 Formal Semantics and Development of MetaML

The study of the formal semantics of MetaML is still ongoing research. In this section, we will present the type system of MetaML [48], and outline a proof of its soundness using a simplified adaptation of the proofs appearing in [29]. The reader is encourage to consult these sources for more detailed treatment of how these results where achieved.

11.1 Big-step Semantics

The syntax of the core subset of MetaML is as follows:

$$e := i \mid x \mid e\,e \mid \lambda x.e \mid \langle e \rangle \mid \sim e \mid \mathsf{run}\ e$$

Values. Values are a subset of terms, which denote the results of computations. Because of the relative nature of Brackets and Escapes, it is important to use a family of sets for values, indexed by the level of the term, rather than just one set. Values are defined as follows:

$$
\begin{array}{lll}
v^0 & \in V^0 & := i \mid x \mid \lambda x.e \mid \langle v^1 \rangle \\
v^1 & \in V^1 & := i \mid x \mid v^1\, v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1 \\
v^{n+2} & \in V^{n+2} & := i \mid x \mid v^{n+2}\, v^{n+2} \mid \lambda x.v^{n+2} \mid \langle v^{n+3} \rangle \mid \tilde{}\, v^{n+1} \mid \mathsf{run}\ v^{n+2}
\end{array}
$$

The set of values has three notable points. First, values can be Bracketed expressions. This means that computations can return pieces of code representing other programs. Second, values can contain applications (inside Brackets) such as $(\lambda y.y)\ (\lambda x.x) \in V^1$. Third, there are no level 1 Escapes in values.

The definition of substitution is standard and is denoted by $e[x := v]$ for the substitution of v for the free occurrences of x in e. The core subset of MetaML can be assigned a big-step semantics as follows [29]:

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x := v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad i \overset{n}{\hookrightarrow} i \qquad x \overset{n+1}{\hookrightarrow} x$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2} \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x.e \overset{n+1}{\hookrightarrow} \lambda x.v}$$

$$\frac{e \overset{0}{\hookrightarrow} \langle v' \rangle \quad v' \overset{0}{\hookrightarrow} v}{\mathsf{run}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\tilde{}\, e \overset{1}{\hookrightarrow} v}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathsf{run}\ e \overset{n+1}{\hookrightarrow} \mathsf{run}\ v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\tilde{}\, e \overset{n+2}{\hookrightarrow} \tilde{}\, v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle}$$

The big-step semantics at level n ($e \overset{n}{\hookrightarrow} v$) always returns a value $v \in V^n$. The index on v is left implicit in the above semantics for clarity.

11.2 Type System

The judgement $\Delta \overset{n}{\vdash} e : \tau, r$ is read *"under the type environment Δ, at level n and syntactically surrounded by r occurrences of Run, the term e has type τ."* The type assignment Δ maps variables to a triple. This triple consists of the type, the level, and the number of surrounding occurrences of Run at the point where this variable was bound (See **Abs** rule).

Going Wrong There are three main kinds of errors related to staging annotations that can occur at run-time:

(1) A variable is used at a level less than the level of the lambda-abstraction in which it is bound, or

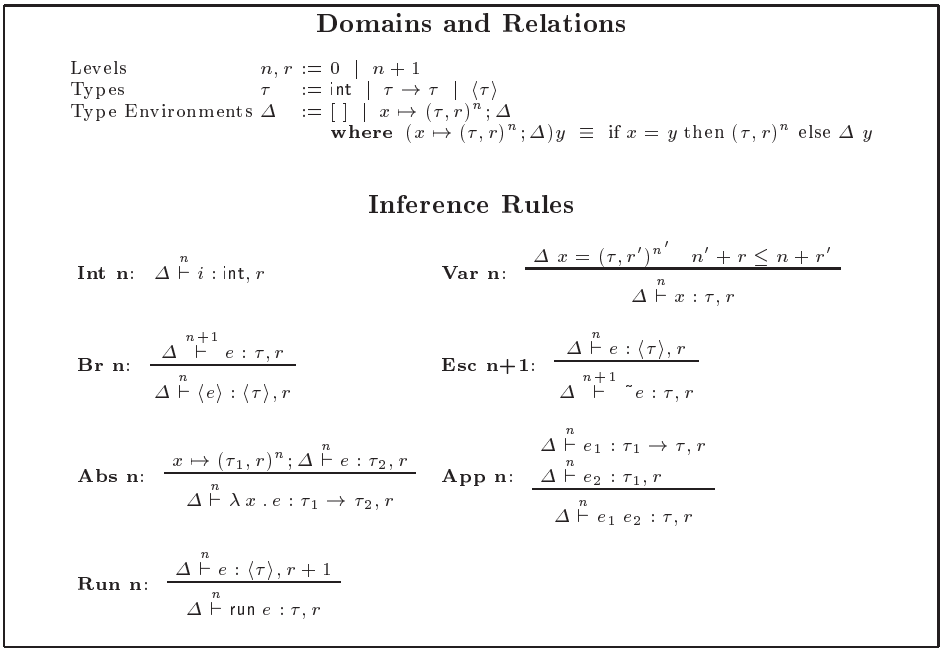

Fig. 1. Type System

(2) Run or Escape are passed values having a non-*code* type, or
(3) Run alters the level of its argument, and can therefore lead to a type (1) error.

The first kind of error is checked by the **Var** rules. Let us assume that our program contains no Run annotations, then r is always zero. Having a rule for $n' \leq n$ allows cross-stage persistence: Variables available in the current stage (n') can be used in all future stages (n). The second kind of error is checked by the **Run n** and **Esc n+1** rules. Detecting the third kind of error is more difficult problem, and is accomplished by keeping track of surrounding occurrences of Run and comparing it to surrounding (uncancelled) Brackets. In essence, assuming the type is correct, we only allow Run, where it removes explicitly manifest Brackets. This is incorporated into the variable rule using the condition $n' + r \leq n + r'$ which ensures that every occurrence of a variable has strictly more surrounding Brackets than Runs. Without this condition we would wrongly allow the program

$$\langle \mathsf{fn\ x} \Rightarrow \tilde{}(\mathsf{run}\ \langle \mathsf{x} \rangle) \rangle$$

which reduces to the term $\langle \mathsf{fn\ x} \Rightarrow \tilde{}\mathsf{x} \rangle$ which is neither a value nor can be reduced any further. In general, this means that we have to be careful with open pieces of code. Specifically, we have to make sure that if Run is applied to an open piece of code, the level of the free variables used in this piece of code will not drop below the level at which they are bound.

For the standard part of the language, code is a normal type constructor that needs no special treatment, and the level n is never changed by the other language constructs.

11.3 Type Preservation

As is common in type preservation proofs, one must prove a Substitution lemma. In addition, because our semantics is also expected to respect the notion of level, we also prove so called Promotion and Demotion lemmas:

Lemma 1 (Level Properties). *The type system has the following three important properties:*

- *Promotion:* $\Delta_1, \Delta_2 \overset{n}{\vdash} e : \tau, r$ *implies* $\Delta_1, \Delta_2^{+(c+d,c)} \overset{n+c+d}{\vdash} e : \tau, r + c$
- *Flex:* $x \mapsto (\tau', r' + 1)^{n'+1}; \Delta_1 \overset{n}{\vdash} e : \tau, r$ *implies* $x \mapsto (\tau', r')^{n'}; \Delta_1 \overset{n}{\vdash} e : \tau, r$
- *Demotion:* $v \in V_{n+1}$ *and* $\Delta \overset{n+1}{\vdash} v : \tau, r + 1$ *implies* $\Delta \overset{n}{\vdash} v : \tau, r$

where $\Delta^{+(c,d)}\ x = (\tau, r + d)^{(n+c)}$ *whenever* $\Delta\ x = (\tau, r)^n$.

Proof. All three properties are proved by straight forward induction over the first typing derivation. The proof of Demotion uses Flex in the case of Abstraction, and takes advantage of the definition of values in the case of Escape to show that Escape at level 1 is not relevant.

Lemma 2 (Substitution). *Let* $r' \leq r$. *Then,* $\Delta' \overset{n'}{\vdash} e' : \tau', r'$ *and* $x \mapsto (\tau', r')^{n'}; \Delta \overset{n}{\vdash} e : \tau, r$ *implies* $\Delta'; \Delta \overset{n}{\vdash} e[x := e'] : \tau, r$

Proof. By straight forward induction over the height of the second typing derivation. The (non-trivial) Variable case uses promotion and takes advantage of the condition that $r' \leq r$.

and we can now prove our main theorem:

Theorem 1 (Type Preservation). *If* $\Delta^{+(1,0)} \overset{n}{\vdash} e : \tau, r$ *and* $e \overset{n}{\hookrightarrow} v$ *then* $v \in V_n$ *and* $\Delta^{+(1,0)} \overset{n}{\vdash} v : \tau, r$.

Proof. By straight forward induction over the height of the evaluation derivation. Application at level 0 uses substitution, and Run at level 0 uses demotion.

11.4 Cross-Stage Persistence

Monolithic Variables Cross-stage persistence can be relaxed by allowing variables to be available at exactly one stage. This seems to be the case in all multi-stage languages known to us to date [34, 7, 13, 9, 10, 6]. Intuitively, they use the following *monolithic* rule for variables (assume $r = 0$):

$$\textbf{Var (Monolithic):}\quad \frac{\Delta\ x = \tau^{n'}}{\Delta \overset{n}{\vdash} x : \tau} \quad \text{when } n' = n$$

We allow the more general condition $n' \leq n$, so an expression like

$$\mathsf{val\ lift\_like} = \mathsf{fn\ x} \Rightarrow \langle \mathsf{x} \rangle$$

is accepted, because inside the Brackets, $n = 1$, and $\Delta\ \mathsf{x} = \alpha^0$. This expression is not accepted by the monolithic variable rule. Note that while the whole function has type $\alpha \to \langle \alpha \rangle$ it does not provide us with the functionality of Lift, because the result of applying lift_like to any value always returns the constant $\langle \mathsf{\%x} \rangle$, not a literal expression denoting the value. This distinction can only be seen at the level of the implementation semantics (discussed below) but not the big-step semantics (discussed above).

The type system rejects the expression

$$\mathsf{fn\ a} \Rightarrow \langle \mathsf{fn\ b} \Rightarrow \tilde{}\mathsf{(a+b)} \rangle$$

because, inside the Escape, $n = 0$, and $(\Delta\ \mathsf{b}) = \alpha^1$, but $1 \not\leq 0$.

12 Limitations to the Expressivity of Run

The type system presented above does not admit the lambda-abstraction of Run. This was, to a large extent, a design choice and a compromise. In particular, if a Run function is introduced into the language as a constant, it breaks the safety of the type system. In this section, we discuss two expressivity problems that arise from this design choice, and how they are addressed in the current implementation.

12.1 Typing Top-Level Bindings

Problem. A MetaML program consists of a sequence of top-level declarations binding variables to terms, followed by a term:

$$\mathit{program} := \ e \ \mid \ \mathsf{val\ x} = e\ ;\ \mathit{program}$$

If we interpret a top-level declaration $\mathsf{val}\ a{=}e_1\ ;\ e_2$ as the application $(\lambda a.e_2)\ e_1$, then we are in the inconvenient situation where we cannot bind a value at top-level that we will eventually want to Run, even if it might otherwise be safe to Run it. This is because $\lambda a.\mathsf{run}\ a$ is not typable in the type system presented in this paper. Thus, this interpretation of $\mathsf{val\ a} = \langle 1 \rangle;\ \mathsf{run\ a}$ would be untypable.

Observations. Top-level bindings have a number of important properties which other (λ-bound) bindings may not: First, every top-level binding is at level 0. Second, all top-level bindings are only within the scope of other top-level bindings. Furthermore, no top-level binding is in the scope of a variable bound at a level greater than 0. This is because, syntactically, no top-level binding can occur in the scope of a piece of code with free variables. Without such free variables, Run does not cause a problem. One of the purposes of the type system was to throw away programs where Run was applied to code with free variables that can cause the computation to get stuck (type (3) errors). Because syntactic restrictions guarantee that variables bound at top-level cannot cause this problem, we use a different type rule for top-level bindings, allowing more safe terms to be typable.

A Solution. The current implementation avoids this problem in MetaML by using the following rule for top-level bindings in the interactive loop:

$$\textbf{Top}: \quad \frac{a \mapsto (\tau_1, r+h)^0; \Delta \overset{0}{\vdash} e_2 : \tau, r \qquad \Delta \overset{0}{\vdash} e_1 : \tau_1, r+h}{\Delta \overset{0}{\vdash} \mathsf{val}\ a{=}e_1\ ;\ e2 : \tau, r}$$

For top-level declarations, the system prints the type of binding as it is entered by the user. Note, however, that h is not printed. In theory, h is existentially quantified in the rule above. In practice, a large number is used. Intuitively, the large h corresponds to the ability to Run values declared at top-level as many times as we want.

Soundness of Top Rule. A let-expression $\mathsf{let}\ a{=}e_1\ \mathsf{in}\ e_2$ is usually interpreted as having the same operational semantics as $(\lambda a.e_2)e_1$. This interpretation can be used to derive the typing rule for let by collecting the simplified assumptions:

$$\cfrac{\cfrac{\cfrac{a \mapsto (\tau_1, r)^n; \Delta \overset{n}{\vdash} e : \tau, r}{\Delta \overset{n}{\vdash} \lambda a.e_2 : \tau_1 \to \tau, r}(\text{Lam}) \qquad \Delta \overset{n}{\vdash} e_1 : \tau_1, r}{\Delta \overset{n}{\vdash} (\lambda a.e_2)e_1 : \tau, r}(\text{App})}{\Delta \overset{n}{\vdash} \mathsf{let}\ a{=}e_1\ \mathsf{in}\ e_2 : \tau, r}(\text{By def.})$$

$$\Downarrow$$

$$\frac{a \mapsto (\tau_1, r)^n; \Delta \overset{n}{\vdash} e : \tau, r \qquad \Delta \overset{n}{\vdash} e_1 : \tau_1, r}{\Delta \overset{n}{\vdash} \mathsf{let}\ a{=}e_1\ \mathsf{in}\ e_2 : \tau, r}(\text{Let})$$

The Top rule is based on an equivalent but non-standard operational interpretation of the declaration $\mathsf{val}\ a{=}e_1\ ;\ e_2$, namely, $\mathsf{run}^{(h)}\ ((\lambda a.\langle^{(h)} e_2\rangle)e_1)$ where h

is the number of repeated occurrences of the construct that it appears as a superscript of. This interpretation is motivated by the fact that if this term is typable, and $e_1 \overset{0}{\hookrightarrow} v_1$, then all the terms in the relation $\mathsf{run}^{(h)}\ ((\lambda a.\langle^{(h)} e_2\rangle)e_1) \overset{0}{\hookrightarrow} e_2[v_1/a]$ are typable whenever the derivation exists. We don't perform the substitution during type-checking, but rather, using the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{a \mapsto (\tau_1, r+h)^n; \Delta \overset{n}{\vdash} e_2 : \tau, r}{a \mapsto (\tau_1, r+h)^n; \Delta \overset{n+h}{\vdash} e_2 : \tau, r+h}(\text{Promotion Lemma})}{a \mapsto (\tau_1, r+h)^n; \Delta \overset{n}{\vdash} \langle^{(h)} e_2\rangle : \langle^{(h)}\tau\rangle, r+h}(\text{Bra } h)}{\Delta \overset{n}{\vdash} \lambda a.\langle^{(h)} e_2\rangle : \tau_1 \to \langle^{(h)}\tau\rangle, r+h}(\text{Lam}) \qquad \Delta \overset{n}{\vdash} e_1 : \tau_1, r+h}{\Delta \overset{n}{\vdash} ((\lambda a.\langle^{(h)} e_2\rangle)e_1) : \langle^{(h)}\tau\rangle, r+h}(\text{App})}{\Delta \overset{n}{\vdash} \mathsf{run}^{(h)}\ ((\lambda a.\langle^{(h)} e_2\rangle)e_1) : \tau, r}(\text{Run } h)}{\Delta \overset{n}{\vdash} \mathsf{val}\ a{=}e_1\ ;\ e_2 : \tau, r}(\text{Def})$$

we arrive at the rule for Top, by collecting the assumptions at the top of the tree of this derivation, and setting n to 0.

Picking a large h works because the Promotion Lemma tells us that if there is an n for which type-checking the top-level let-binding is possible, then it will also be possible for all $n' > n$.

12.2 Use of Run inside Functions

It would be useful if the type system allowed us to express functions such as the following:

```
val f = fn x : int list ⇒ run (gen x);
```

This is a function that takes a list of integers, generates an intermediate program based on the list, and then executes the generated program. The type system for the core language does not admit this term (for any previously declared variable gen). In our experience, most such functions where quite small, and we could often achieve the same effect as f e by taking advantage of the power of the let-rule described above:

```
val a = gen e;
val r = run a;
```

This trick is useful, but is not satisfactory from the point of view of the modularity of the code, as it forces us to do a kind of "inlining" of f to get around the type system. We conjecture that it is possible to relax the type system somewhat using rules such as

$$\textbf{Run n:}\quad \dfrac{x_i \mapsto (b_i, r_i+1)^0; \Delta \overset{n}{\vdash} e : \langle\tau\rangle, r+1}{x_i \mapsto (b_i, r_i)^0; \Delta \overset{n}{\vdash} \mathsf{run}\ e : \tau, r}$$

where b is a base-type (such as int or —int list). Intuitively, this typing rule assures us that whenever a basic value (that is, not involving code) is available at level 0, it can be used in a context with as many surrounding occurrences of Run as needed. This rule would allow us to type the expression above. However, it is still *ad hoc*, and we hope to formulate a more systematic basis for such rules in future work.

13 Implementation Semantics

The big-step semantics presented above does not capture all the relevant operational details addressed in the implementation of MetaML. The three main exceptions are the need for 1) distinguishing between real and symbolic bindings, 2) run-time generation of names (*gensym*), and 3) cross-stage persistent constants. In this section we present a semantics which describes our implementation. While we have worked hard to keep our implementation both efficient and faithful to the big-step semantics, the formal proof of their relation is still ongoing work.

13.1 Real and Symbolic Binds, *gensym*, and Cross-Stage Constants

The implementation semantics consists of rules for *reduction* $\Gamma \vdash e \hookrightarrow v$, essentially applying the Application and Run rules, and *rebuilding* $\Gamma \vdash e \overset{n+1}{\hookrightarrow} e$, indexed by a level $n+1$, essentially constructing code while evaluating Escaped computations inside Brackets, where the environment Γ binds a variable to a value. Reduction is standard for the most part. A subtlety relating to variable binding causes a problem that makes environments somewhat complicated. In particular, some variables are not yet bound when rebuilding is taking place. For example, rebuilding the term $\langle \mathsf{fn\ x} \Rightarrow \tilde{}(\mathsf{id}\ \langle \mathsf{x} \rangle) \rangle$ requires reducing the application $\mathsf{id}\ \langle \mathsf{x} \rangle$. But while reducing this application, the variable x is not yet bound to a value. In the intended semantics of MetaML, we really want this variable to be simply a name that is not susceptible to accidental name capture at run-time.

To solve this problem, bindings in environments come in two flavors: real ($\mathsf{Real}(v)$) and symbolic ($\mathsf{Sym}(x)$). The extension of the environment with real values occurs only in the rule **App 0**. Such values are returned (**Var 0**) under reduction, or injected into constant %_ terms (**RVar n+1**) under rebuilding. In essence, the three tags Real(_), Sym(_) and %_ work together to provide us with the set of coercions needed to deal with free variables and to implement cross-stage persistence.

Another feature of the implementation semantics is that it is self-contained, in that it does not use a substitution operation. Instead, substitution is performed by the rebuilding operation. In particular, in the absence of staging annotations rebuilding is just capture-free substitution of symbolic variables bound in Γ.

Rebuilding is used in two rules, **Abs 0** where it is used for capture-free substitution, and **Bracket 0** where it is applied to terms inside Brackets and it describes how the delayed computations inside a value are constructed.

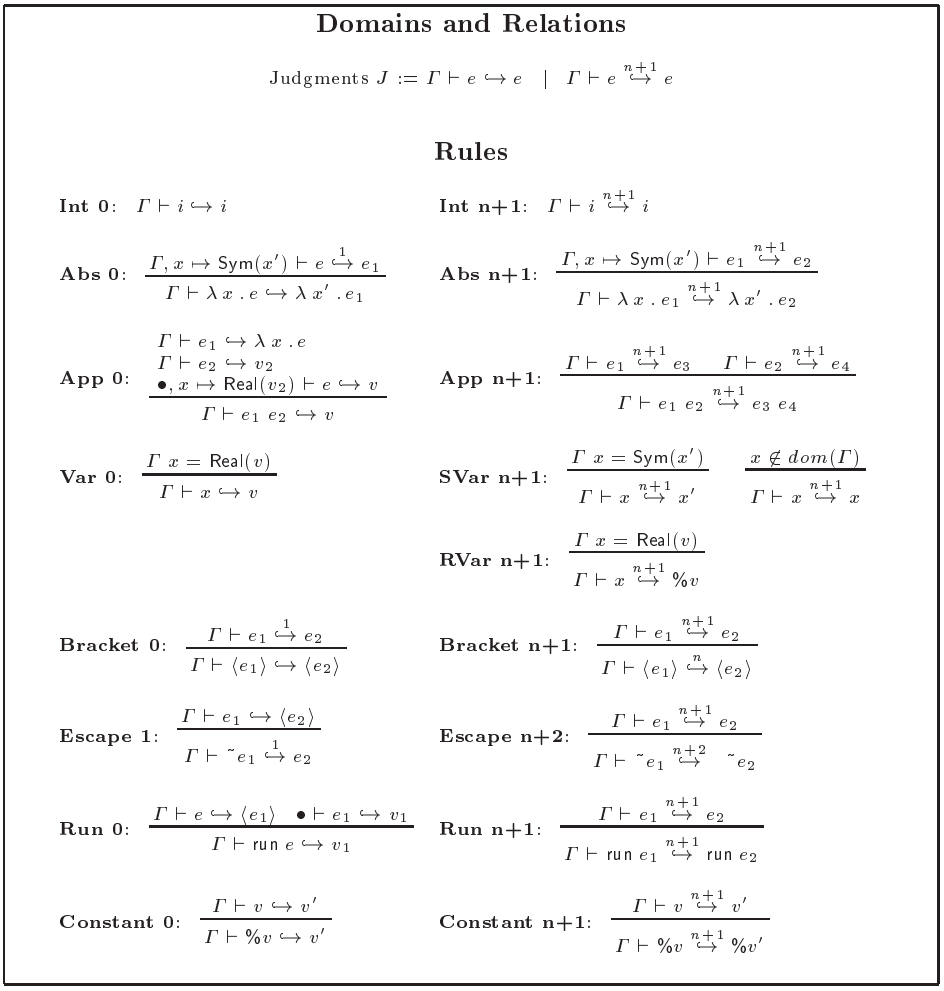

Fig. 2. Implementation Semantics

The type system ensures that in rule **Abs 0**, there are no embedded Escapes at level 1 that will be encountered by the rebuilding process, so the use of rebuilding in this rule implements nothing more than capture-free substitution.

In the rebuilding rule **Escape 1**, an Escaped expression at level 1 indicates a computation must produce a code-valued result $\langle e_2 \rangle$, and rebuilding returns the term e_2. The role of n in the judgement is to keep track of the level of the expression being built. The *level* of a subexpression is the number of uncancelled surrounding Brackets. One surrounding Escape cancels one surrounding Bracket. Hence, n is incremented for an expression inside Brackets (**Bracket**), and decremented for one inside an Escape (**Escape**). Note that there is no rule for Escape at level 0: Escape must appear inside uncancelled Brackets.

The reduction rule **Bracket 0** describes how a code value is constructed from a Bracketed term $\langle e_1 \rangle$. The embedded expression is stripped from its Brackets, rebuilt at level 1, and the result of this rebuilding is then rewrapped with Brackets.

So to summarize, altogether rebuilding has three distinct roles:

1. To replace all known variables with a constant expression ($\%v$) where the v comes from $\mathsf{Real}(v)$ bindings in Γ (rule **RVar n+1**).
2. To rename all bound variables. Symbolic $\mathsf{Sym}(x')$ bindings occur in rules **Abs 0** and **Abs n+1** where a term is rebuilt, and new names are introduced to avoid potential variable capture. These new names are projected from the environment (rule **SVar n+1**).
3. To execute Escaped expressions to obtain code to "splice" into the context where the Escaped term occurs (rule **Escape 1**).

The reduction rule **Run 0** describes how a code-valued term is executed. The term is reduced to a code-valued term, and the embedded term is then reduced in the empty environment to produce the answer. The empty environment is sufficient because cross-stage persistent free variables in the original code-valued term have been replaced by constant expressions ($\%v$), and all free variables are handled by the idempotent case in **SVar n+1**.

13.2 The Notion of a Stage

In the introduction, we gave the intuitive explanation for a stage. After presenting the semantics for MetaML, we can now provide a more formal definition. We define (the trace of) *a stage* as *the derivation tree generated by the invocation of the derivation* $\bullet \vdash e_1 \hookrightarrow v_1$ (cf. **Run 0** rule). Note that while the notion of a level is defined with respect to *syntax*, the notion of a stage is defined with respect to *a trace of an operational semantics*. Although quite intuitive, this distinction was not always clear to us, especially that there does not seem to be any comparable definition in the literature with respect to an operational semantics.

The levels of the subterms of a program and the stages involved in the execution of the program can be unrelated. A program $\langle 1{+}\mathsf{run}\ \langle 4{+}2 \rangle \rangle$ has expressions at levels 0, 1, and 2. If we define the "level of a program" as the maximum level of any of its subexpressions, then this is a 2-level program. The evaluation of this expression (which just involves rebuilding it), involves no derivations $\bullet \vdash e_1 \hookrightarrow v_1$. On the other hand, the evaluation of slightly modified 2-level program $\mathsf{run}\ \langle 1{+}\mathsf{run}\ \langle 4{+}2 \rangle \rangle$ involves two stages.

To further illustrate the distinction between levels and stages, let us define the "number of stages" of a program as the number of times the $\bullet \vdash e_1 \hookrightarrow v_1$ is used in the derivation involved in its evaluation. Consider:

$$(\mathsf{fn\ x} \Rightarrow \mathsf{if\ P\ then\ x\ else\ lift(run\ x)})\ \langle 1{+}2 \rangle$$

where P is an arbitrary problem. The number of stages in this program is not statically decidable. Furthermore, we cannot say, in general, which occurrence of

Run will be ultimately responsible for triggering the computation of the addition in expression ⟨1+2⟩.

Recognizing this mismatch was a useful step towards finding a type-system for MetaML, which employs the static notion of level to approximate the dynamic notion of stage.

13.3 Why is Lambda-Abstraction not Enough?

It may appear that staging requires only lambda-abstraction, and its dual operation, application. While this may be true for certain applications, for the domain of program generation there are two additional capabilities that are needed: First, a delayed computation must maintain an intensional representation, so that users can inspect the code produced by their generators, and so that it can be printed and fed into compilers. In a compiled implementation, lambda-abstractions lose their high-level intensional representation, and neither of these is possible.

Second, generators often need to perform "evaluation under lambda". This is necessary for almost any staged application that performs some kind of unfolding, and is used in functions like back. Although we cannot prove it, the effect of Escape (under lambda) cannot be imitated in the call-by-value λ-calculus without extending it with additional constructs. To further explain this point, we will show an example of the result of encoding of the operational semantics of MetaML in SML/NJ.

The essential ingredients of a program that requires more than abstraction and application for staging are Brackets, dynamic (non-level 0) abstractions, and Escapes. Lambda-abstraction over unit can be used to encode Bracket, and application to unit to encode Run. However, Escape is considerably more difficult. In particular, the expression inside an Escape has to be executed before the surrounding delayed computation (closure) is constructed. This becomes a problem when variables introduced inside the delayed expression occur in the Escaped expression. For example: ⟨fn x ⇒ ~(f ⟨x⟩)⟩.

One way to imitate this behavior uses two non-pure SML features. References can be used to simulate evaluation under lambda, and exceptions to simulate the creation of uninitialized reference cells. Consider the following sequence of MetaML declarations:

```
fun G f = ⟨fn x ⇒ ~(f ⟨x⟩)⟩
val pc = G (fn xc ⇒ ⟨(~xc,~xc)⟩)
val p5 = (run pc) 5
```

The corresponding imitation in SML would be:

```
exception not_yet_defined
val undefined = (fn () ⇒ (raise not_yet_defined))
fun G f =
    let val xh = ref undefined
        val xc = fn () ⇒ !xh ()
```

```
        val nc = f xc
    in
      fn () ⇒ fn x ⇒ (xh:=(fn () ⇒ x);nc ())
    end;
val pc = G (fn xc ⇒ fn () ⇒ (xc(),xc()))
val p5 = (pc ()) 5
```

In this translation, values of type $\langle\alpha\rangle$ are encoded by delayed computations of type $unit \to \alpha$. We begin by assigning a lifted undefined value to undefined. Now we are ready to write the analog of the function G. Given a function f, the function G first creates an uninitialized reference cell xh. This reference cell corresponds to the occurrences of x in the application f ⟨x⟩ in the MetaML definition of G. Intuitively, the fact that xh is uninitialized corresponds to the fact that x will not yet be bound to a fixed value when the application f ⟨x⟩ is to be performed. This facility is very important in MetaML, as it allows us to unfold functions like f on "dummy" variables like x. The expression fn () ⇒ !xh () is a delayed lookup of xh. This corresponds to the Brackets surrounding x in the expression f ⟨x⟩. Now, we simply perform the application of the function f to this delayed construction. It is important to note here that we are applying f as it is passed to the function G, before we know what value x is bound to. Finally, the body of the function G returns a delayed lambda-abstraction, which first assigns a delayed version of x to xh, and then simply includes an applied ("Escaped") version of nc in the body of this abstraction.

The transliteration illustrates the advantage of using MetaML rather than trying to encode multi-stage programs using lambda-abstractions, references, and exceptions. The MetaML version is shorter, more concise, looks like the unstaged version, and is easier to understand.

One might consider an implementation of MetaML based on this approach, hidden under some syntactic sugar to alleviate the disadvantages listed above. The lambda-delay method has the advantage of being simply a machine-independent manipulation of lambda-terms. Unfortunately it fails to meet the intensional representation criterion, and also incurs some overhead not (necessarily) incurred in the MetaML version. In particular, the last assignment to the reference xh is delayed, and must be repeated every time the function returned by G is used. The same happens with the application ("Escaping") of nc. Neither of these expenses would be incurred by the MetaML version of G. Intuitively, these operations are being used to connect the meta-level variable x to its corresponding object-level xh. In MetaML, these overheads would be incurred exactly once during the evaluation of run pc as opposed to every time the function resulting from pc () is applied.

14 Optimization of Generated Code

While the semantics presented above is sufficient for executing MetaML programs, code generated by such programs would contain some superfluous computations. Not only can these superfluous computations make it more costly to

execute the generated programs, but it can also make the code larger, and hence harder for humans to understand. In what follows, we discuss two such kinds of computations, and how we deal with these problems in the implementation of MetaML.

14.1 Safe Beta Reduction

Consider the following example:

```
val g = ⟨fn x ⇒ x * 5⟩;
val h = ⟨fn x ⇒ (~g x) - 2⟩;
```

If we use the big-step semantics presented above, the variable `h` evaluates to ⟨`fn d1 ⇒ ((fn d2 ⇒ d2 * 5) d1) - 2`⟩. MetaML actually returns ⟨`fn d1 ⇒ (d1 * 5) - 2`⟩ because it attempts to perform a safe beta reduction whenever a piece is code is Escaped into another one. A beta reduction is *safe* if it does not affect termination properties. There is one safe case which is particularly easy to recognize: An application of a lambda-abstraction to a constant or a variable can always be symbolically reduced without affecting termination. This is justified because the β rule is expected to hold at all levels. Performing a safe beta step does not change the termination or the order of evaluation of the program so it is performed once when the code is built rather than repeatedly when the code is Run.

14.2 Nested Escapes

Consider the case where a deeply Bracketed term e at level n is Escaped all the way to level 0. In order to execute this term (which Escapes to level 0) it must be rebuilt n times. Consider the reduction sequence sketched below for the term `run (run ⟨⟨ ~~e⟩⟩)`, where `e` is bound in Γ to ⟨`5`⟩, of which we show only the innermost Run.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{e \hookrightarrow \langle\langle 5\rangle\rangle}{\sim e \overset{1}{\hookrightarrow} \langle 5\rangle}}{\sim\sim e \overset{2}{\hookrightarrow} \sim\langle 5\rangle}}{\langle\sim\sim e\rangle \overset{1}{\hookrightarrow} \langle\sim\langle 5\rangle\rangle}}{\langle\langle\sim\sim e\rangle\rangle \hookrightarrow \langle\langle\sim\langle 5\rangle\rangle\rangle} \quad \dfrac{\dfrac{\dfrac{5 \overset{1}{\hookrightarrow} 5}{\langle 5\rangle \hookrightarrow \langle 5\rangle}}{\sim\langle 5\rangle \overset{1}{\hookrightarrow} 5}}{\langle\sim\langle 5\rangle\rangle \hookrightarrow \langle 5\rangle}}{\mathsf{run}\ \langle\langle\sim\sim e\rangle\rangle \hookrightarrow \langle 5\rangle}$$

The term ⟨5⟩ is rebuilt two times. A simple refinement can prevent this from happening. We change the rebuilding of Escaped expressions at levels greater than 1 by adding the rule **Escape Opt n+2** in addition to the rule **Escape n+2**.

$$\textbf{Escape Opt n+2:}\quad \frac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} \langle e_2\rangle}{\Gamma \vdash \sim e_1 \overset{n+2}{\hookrightarrow} e_2}$$

$$\textbf{Escape n+2:}\quad \frac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} e_2}{\Gamma \vdash \tilde{}e_1 \overset{n+2}{\hookrightarrow} \ \tilde{}e_2}$$

Thus a long sequence of Escapes surrounded by an equal number of Brackets gets rebuilt exactly once. This optimization is justified because rebuilding more than once performs no useful work.

Note that these optimization eliminate some redexes that the user might expect to see in the generated code, and hence make it hard to understand *why* a particular program was generated. In our experience, the resulting smaller, simpler programs, are easier to understand and seemed to make the optimizations worthwhile.

15 Discussion and Related Works

Nielson and Nielson pioneered the investigation of multi-level languages with their work on two-level functional languages [30, 34, 31, 32]. They have developed an extensive theory for the denotational semantics of two-level languages, including a framework for abstract interpretation [33]. The framework developed is for a general ("B-level") language, where B is an arbitrary, possibly partially-ordered set. Recently, Nielson and Nielson proposed an algebraic framework for the specification of multi-level type systems [35, 36].

Gomard and Jones [11] use a statically-typed two-level language for partial evaluation of the untyped λ-calculus. This language is the basis for many BTAs. The language allows the treatment of expressions containing monolithic free variables. They use a "const" construct only for constants of ground type. Our treatment of variables in the implementation semantics is inspired by their work.

Glück and Jørgensen [9] present the novel idea of multi-level BTA (MBTA), as an efficient and effective alternate to multiple self-application. An untyped multi-level language based on Scheme is used for the presentation. MetaML has fewer primitives than this language, and our focus is more on program generation issues rather than those of BTA. It is also worth noting that all intermediate results in their work are printable, that is, have a high-level intensional representation. In MetaML, cross-stage persistence allows us to have intermediate results (between stages) that contain constants for which no intentional representation is available. While this is very convenient for run-time code generation, it made the proper specification of MetaML more difficult. For example, we can't use their "Generic Code Generation functions" to define the language. A second paper by Glück and Jørgensen [10] demonstrates the impressive efficiency of MBTA, and the use of constraint-solving methods to perform the analysis. The MBTA is type-based, but underlying language is not statically typed.

Thiemann [50] studies a two-level language with eval, apply, and call/cc in the context of the partial evaluation of a larger subset of scheme than had been previously studied. A BTA based on constraint-solving is presented. Although

the problems with eval and call/cc are highlighted, a different notion of types is used, and the complexity of introducing eval into a multi-stage language does not manifest itself. Thiemann also deals with the issue of variable-arity functions, a practical problem when dealing with eval in Scheme.

Hatcliff and Glück studied a multi-stage flow-chart language called S-Graph-n, and thoroughly investigated the issues involved in the implementation of such a language [13]. The syntax of S-Graph-n explicitly captures all the information necessary for specifying the staging of a computation: each construct is annotated with a number indicating the stage during which it is to be executed, and all variables are annotated with a number indicating the stage of their availability. S-Graph-n is not statically typed, and the syntax and formal semantics of the language are quite sizable. Programming in S-Graph-n requires the user to annotate every construct and variable with stage annotations, and ensuring the consistency of the annotations is the user's responsibility. In their work, Hatcliff and Glück identified *language-independence* of the internal representation of "code" as an important characteristic of any multi-stage language.

Sheard and Nelson investigate a two-stage language for the purpose of program generation [43]. The base language was statically typed, and dependent types were used to generate a wider class of programs than is possible by MetaML restricted to two stages. Sheard and Shields [44] investigate a dynamic type systems for multi-staged programs where some type obligations of staged computations can be put off till run-time.

Davies and Pfenning present a statically-typed multi-stage language Mini-ML$^{\Box}$, motivated by constructive modal logic [6]. A formal proof is presented for the equivalence of binding-time correctness and modal correctness. MetaML type-system was motivated primarily by operational considerations. Their language has two constructs, box and let-box, which correspond roughly to Brackets and Run. Mini-ML$^{\Box}$'s $\Box$ type constructor is similar to *code*. Mini-ML$^{\Box}$ can simulate Lift, but a stage-zero function, for example, cannot be made persistent. Finally, functions like back are not expressible in Mini-ML$^{\Box}$.

The multi-stage language Mini-ML$^{\bigcirc}$ [6] is motivated by a linear-time constructive modal logic. The language allows staged expressions to contain monolithic free variables. The two constructs of Mini-ML$^{\bigcirc}$; next and prev, correspond quite closely to MetaML's Brackets and Escape. The type constructor $\bigcirc$ also corresponds roughly to *code*. Unfortunately, eval is no longer expressible in the language.

Moggi advocates a categoric approach to two-level languages [28]. He also points out that the use of stateful functions such as *gensym* or *newname* in the semantics makes their use for formal reasoning hard. The implementation semantics presented in this paper uses a *gensym*, but the big-step semantics does not.

Figure 3 is a summary of the distinguishing characteristics of some of the languages discussed here. For Levels, "2" mean it is a two-level language, and "+" means multi-level. For static typing, "1" means only first level is statically checked.

Facility	**Example**	Nielson & Nielson [34]	Gomard & Jones [11]	Glück & Jørgensen [9]	Thiemann [50]	Hatcliff & Glück [13]	$\lambda^{\Box}$ [7]	$\lambda^{\bigcirc}$ [6]	$\lambda^{\mathcal{M}}$
Levels	$\langle \lambda x.x \rangle$	2	2	+	2	+	+	+	+
Static Typing		Y	1	N	N	N	Y	Y	Y
Monolithic Var.	$\langle \lambda x.\tilde{}(f\ \langle x \rangle) \rangle$	Y	Y	Y	Y	Y	N	Y	Y
Reflection	Run or eval	N	N	N	Y	N	Y	N	Y
Persistence	$\lambda f.\langle \lambda x.f\ x \rangle$	N	N	N	N	N	N	N	Y
Portability		Y	Y	Y	Y	Y	Y	Y	N

Fig. 3. Comparative feature set

16 Ongoing Work and Open Questions

The work reported in this paper has directed our attention to many important questions relating to multi-stage computation in general, and MetaML in particular. We are currently investigating a number of aspects of MetaML:

1. A denotational semantics assigns an abstract meaning to a language. We expect that the works of Nielson and Nielson, and Moggi, will serve as a good basis for assigning such a semantics to MetaML.
2. Reduction semantics and equational theory, to serve as a practical basis for formal reasoning about program optimizations and the equivalence of programs. A reduction semantics is investigated in [48], but is limited due to a subtlety with the non-standard definition of substitution.
3. MetaML admits the analog of *polyvariant specialization* [14] by annotating differently copies of the same program. It is not yet clear how to make this task easier for the programmer.
4. Validating the implementation with respect to more abstract formulations of the semantics of MetaML.
5. Extending to effects. The extension of the current type system with effects is not obvious. For example, adding references and sequencing *a la* SML allows the following unsafe program:

   ```
   val r = ref ⟨1⟩;
   val c = ⟨fn x ⇒ ~(r := ⟨x⟩; 2)⟩;
   val i = run (!r);
   ```

6. Providing a more general solution to the let-binding problem. While we have proposed one solution to the let-binding problem at top-level, this solution does not carry over the let-bindings at higher levels.
7. Simplifying the type system. The Flex property suggests that it may be sufficient to keep track only of the difference between n and r in the typing environment. Also, our remedies for the limitation in the expressivity of Run were *ad hoc*.

17 Conclusion

We have described a multi-stage programming language which we call MetaML. MetaML was designed as a programming language. Our primary purpose was to support the writing of multi-stage programs. Because of this our design choices where different from those of other multi-stage systems. We believe that MetaML helps us in understanding and communicating ideas about multi-stage programs, partial evaluation, and the complex process of BTA in much the same way that the boxed/unboxed(#) distinction provides a language for understanding boxing optimizations as source-to-source transformations [21].

This paper identifies a number of language features that we have found to be essential when writing multi-stage programs:

- **Cross-stage persistence.** The ability to use variables from any past stage is crucial to writing staged programs in the manner to which programmers are accustomed. Cross-stage persistence provides a solution to hygienic macros in a typed language, that is macros which bind identifiers in the environment of definition, which are not "captured" in the environment of use.
- **Multi-stage aware type system.** The type checker reports phase errors as well as type errors. This is crucial when debugging multi-stage programs.
- **Display of code.** When debugging, it is important for users to observe the code produced by their programs. This requires a display mechanism (pretty-printer) for values of type code.
- **Display of Constants.** The origin of a cross-stage persistence constant can be hard to identify. The named `%_` tags provide an approximation of where these constants came from. While these tags can sometimes be misleading, they are often quite useful.
- **The connection between $\langle \mathsf{A} \rangle \to \langle \mathsf{B} \rangle$ and $\langle \mathsf{A} \to \mathsf{B} \rangle$.** Having these mediating functions reduces, sometimes drastically, the number of annotations needed to stage programs.
- **Lift.** The Lift annotation makes it possible to force computation in a early stage and Lift this value into a program to be incorporated at a later stage. While it may seem that cross-stage persistence makes Lift unnecessary, Lift helps produce code which is easier to understand, because constants become explicit.
- **Safe beta- and Escape-reduction.** These optimizations improve the generated code, and often make it more readable.

We have built an implementation which was used to program the examples in this paper and other larger examples (cf. [49]). Currently, the implementation supports polymorphic type-inference. We are also extending this implementation to include all the features SML.

Acknowledgments: The research on MetaML, and this paper, have benefited greatly from our collaboration with Zino Benaissa and Eugenio Moggi. Over the years, have have also had the good fortune of getting valuable inputs from Koen Claessen, Olivier Danvy, Rowan Davies, Robert Glück, Jim Hook, Neil

Jones, John Launchbury, Peter Lee, Erik Meijer, Flemming Nielson, Dino Oliva, Frank Pfenning, Amr Sabry, Phil Wadler and the members of PacSoft. A special thanks is due for Lisa Walton on comments on the final manuscript. We would also like to thank the anonymous referees for careful and constructive comments which sharpened our ideas and improved the organization and presentation of the work. Last, but not least, we thank the editor, Charles Consel, for facilitating our communication with the referees during the course of the review of this paper.

References

1. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, January 1993.
2. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.
3. Olivier Danvy. Across the bridge between reflection and partial evaluation. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 83–116. North-Holland, 1988.
4. Olivier Danvy. Type-directed partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 242–257, Florida, January 1996. New York: ACM.
5. Olivier Danvy, Karoline Malmkjaer, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 1(19), 1995.
6. Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings,* 11th *Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.
7. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.
8. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
9. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
10. Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
11. Carsten K. Gomard and Neil D. Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
12. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, The Netherlands, June 1997.

13. John Hatcliff and Robert Glück. Reasoning about hierarchies of online specialization systems. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 161–182. Springer-Verlag, 1996.
14. Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301, Edinburgh, U.K., April 1994. Springer-Verlag.
15. James Hook and Tim Sheard. A semantics of compile-time reflection. Technical Report CSE 93-019, Oregon Graduate Institute, 1993.
16. Luke Hornof, Charles Consel, and Jacques Noyé. Effective specialization of realistic programs via use sensitivity. In *SAS 1997*, pages 293–314, Paris, France, September 1997.
17. Luke Hornof and Jacques Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 63–73, Amsterdam, The Netherlands, 12–13 June 1997.
18. Neil D. Jones. Mix ten years later. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 24–38. New York: ACM, New York: ACM, June 1995.
19. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
20. Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
21. Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, September 91.
22. Richard B. Kieburtz, Francoise Bellegarde, Jef Bell, James Hook, Jeffrey Lewis, Dino Oliva, Tim Sheard, Lisa Walton, and Tong Zhou. Calculating software generators from solution specifications. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 546–560. Springer-Verlag, 1995.
23. Richard B. Kieburtz, Laura McKinney, Jeffrey Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *18th International Conference in Software Engineering*, March 1996.
24. Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU-CS-93-225, Carnegie Mellon University, Dec. 1993.
25. Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.
26. Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. Amphion: Automatic programming for scientific subroutine libraries. *NASA Science Information Systems Newsletter*, 31:22–25, 1994.
27. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
28. Eugenio Moggi. A categorical account of two-level languages. In *MFPS 1997*, 1997.
29. Eugenio Moggi, Walid Taha, Zine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, OGI, October 1998.

30. Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.
31. Flemming Nielson. Correctness of code generation from a two-level meta-language. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP 86)*, volume 213 of *LNCS*, pages 30–40, Saarbrücken, FRG, March 1986. Springer.
32. Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, December 1989.
33. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.
34. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
35. Flemming Nielson and Hanne Riis Nielson. Multi-level lambda-calculi: An algebraic description. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354. Berlin: Springer-Verlag, 1996.
36. Flemming Nielson and Hanne Riis Nielson. A prescriptive framework for designing multi-level lambda-calculi. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, June 1997.
37. Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.
38. Tim Sheard. A user's guide to trpl, a compile-time reflective programming language. Technical Report COINS Tech. Rep. 90-109, Dept. of Computer and Information Science, University of Massachusetts, 1990.
39. Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
40. Tim Sheard. Guide to using crml, cmopile-time reflective ml. (Available from author's home-page), October 1993.
41. Tim Sheard. Type parametric programming. Technical Report CSE 93-018, Oregon Graduate Institute, 1993.
42. Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, June 1997.
43. Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical Report OGI-TR-95-013, Oregon Graduate Institute of Science and Technology, 1995.
44. 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*, January 1998.
45. Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982.
46. Brian Cantwell Smith. Reflection and semantics in lisp. In *ACM Symposium on Principles of Programming Languages*, January 1984.
47. D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson, and S. Alagic. Type-safe linguistic reflection: A generator technology. Technical Report FIDE/92/49, ESPRIT BRA Project 3070 FIDE, 1992.

48. Walid Taha, Zine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th ICALP*, Aalborg, Denmark, July 1998.
49. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.
50. Peter Thiemann. Towards partial evaluation of full Scheme. In Gregor Kiczales, editor, *Reflection 96*, pages 95–106, San Francisco, CA, USA, April 1996.