Multi-Stage Programming with Explicit Annotations

Walid Taha & Tim Sheard
Oregon Graduate Institute of Science & Technology
{walidt,sheard}@cse.ogi.edu *

Abstract

We introduce MetaML, a statically-typed multi-stage programming language extending Nielson and Nielson's two stage notation to an arbitrary number of stages. MetaML extends previous work by introducing *four* distinct staging annotations which generalize those published previously [25, 12, 7, 6]

We give a static semantics in which type checking is done once and for all before the first stage, and a dynamic semantics which introduces a new concept of *cross-stage persistence*, which requires that variables available in any stage are also available in all future stages.

We illustrate that staging is a manual form of binding time analysis. We explain why, even in the presence of automatic binding time analysis, explicit annotations are useful, especially for programs with more than two stages.

A thesis of this paper is that multi-stage languages are useful as programming languages in their own right, and 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, and an extended two-stage example elaborating a number of practical issues.

1 Introduction

Multi-stage languages have recently been proposed as intermediate representations for partial evaluation [12, 9, 10] and runtime code generation [7]. These languages generalize the well-known two-level notation of Nielson & Nielson [25] to an arbitrary number of levels.

A major thesis of this paper is that *multi-stage languages are useful not only as intermediate representations, but also as programming languages in their own right.* Multi-stage programming is important whenever performance is important. But there is very little language support for writing multi-stage programs. This paper extends previous work on multi-stage programming with features that are of *practical use to real programmers.*

We introduce MetaML, a statically-typed multi-stage programming language extending Nielson and Nielson's two-level notation to an arbitrary number of stages (similar to their B-level language). MetaML is an extension of a Hindley-Milner polymorphically-typed [22] call-by-value λ-calculus [13] with support for sums, products, recursion, polymorphism, primitive datatypes and static type-inference. It provides the following extensions not found in previous work on multi-stage systems:

- Four distinct staging annotations, which we believe are necessary and sufficient for all multi-stage programming. (Section 5) These annotations generalize and safely combine those published previously [25, 12, 7, 6]. The formal proof of safety is ongoing work.
- A type system ensuring the well-formedness of acceptable multi-stage programs. Type checking is done once and for all before the first stage (Section 10.1).
- Variables of any stage are available in all future stages. This feature, in a language which also contains `run`[1] makes MetaML's annotations strictly more expressive than the languages of Nielsen & Nielsen [25, 24], Davies & Pfenning [7], and Davies [6]. We also deal with the interesting technical problem of ensuring the hygienic binding of free variables (Section 10.2) in code expressions.
- A non-Hindley-Milner, second-order type judgement for the `run` annotation to ensure that no code is ever `run` in a context in which it is undefined.

As a consequence of the above properties, MetaML provides a programming language suitable for expressing staged computations explicitly. We believe that MetaML can have positive implications for understanding and communicating ideas about multi-stage programs, partial evaluation and the complex process of binding-time analysis in much the same way that the boxed / unboxed(#) distinction provides a language for understanding boxing optimizations as source-to-source transformations [16].

*The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-93-C-0069, and NSF Grant IRI-9625462. **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] An *eval*-like operator.

2 Why Multi-stage Programs?

The concept of a stage arises naturally in a wide variety of situations. For a compiled language, there are two distinct stages: compile-time, and run-time. But 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 object code.

Yet despite the numerous examples of multi-stage software systems, almost all these systems have realized staging in ad-hoc ways. Our goal is to provide a language with well-designed support for multi-staged programming by using explicit staging annotations. In particular, a multi-stage programming language supplies a basis for generation technology. Generators can provide dramatic improvements in the following areas:

Efficiency. Specializing a function on a fixed argument can lead to dramatic efficiency gains. Program generators can provide the same efficiency gains that partial evaluation does.

Productivity and reuse. When a programming task or activity becomes routine, programmers can use program generators to encapsulate their knowledge of the routine task. This capture of a problem family rather than a single problem increases programmer productivity. Program generators let experts capture their knowledge in a clear (and hence reusable) notation that can then be used for synthesising the desired software component [21, 17, 18].

Reliability and quality. The greatest source of errors in code maintenance is human intervention. When less human intervention is needed to modify a software product, there are proportionately fewer opportunities for error insertion and less rework of code is necessary. Automatically generated components require little manual rework after a re-generation.

Our language, MetaML, was designed as basis for an integrated generator system. It provides an approach radically different from, and superior to, the classic "programs-as-strings" view that seems to predominate in many ad-hoc software generation systems. MetaML is tightly integrated in this sense.

3 Relationship to Partial Evaluation

Today, the most sophisticated automatic staging techniques are found in partial evaluation systems [15]. Partial evaluation optimizes a program using a priori information about some of that program's inputs. The goal is to identify and perform as many computations as possible in a program before run-time.

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

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

The relationship between partial-evaluation and multi-stage programming is that the intermediate data structure between the two steps is a *two-stage annotated program* [2], 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.

4 Why Explicit Annotations?

If BTA performs staging automatically, why should programmers stage programs manually? They shouldn't *have* to, but there are several important reasons why they may *want* to:

Pragmatic. While there are advantages to discussing the semantics of annotated programs and the techniques of BTA at the same time, we feel that the complexity of the semantics of annotated programs warrants studying them in (relative) isolation of other partial evaluation issues.

Pedagogical tool. It has been observed that it is sometimes hard for users to understand the workings of partial evaluation systems [14]. New users often lack a good mental model of how partial evaluation systems work. Although BTA is an involved process, requiring special expertise, the annotations it produces are relatively simple and easy to understand. However, 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. 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 [15]. Nielson & Nielson's two-stage notation is the only widely accepted notation for expressing staged computation. But Nielson & Nielson's notation is not widely viewed as a programming language, perhaps because over-bars and under-bars do not appear on the standard keyboard and no implementation of it is in widespread use.

Controlling Evaluation Order. Whenever performance is an issue, control of evaluation order is essential. BTA optimizes the evaluation order, but sometimes it is just easier to say what you want than to force a BTA to discover it. 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 addition, BTA for programs with more than two stages is still imprecise. Hand annotation may be the only feasible mechanism for staging multi-stage programs, and may be the only mechanism expressive enough for the degree of control needed in many circumstances.

High-Level Program Generation. As we will also illustrate in this paper, staging annotations also provide a powerful tool for high-level program generation. No explicit construction of parse trees is needed. As a consequence, generators can be simpler and more reliable than their hand constructed counterparts. It is also easier to verify the correctness of both the generators and the programs they generate, as the issues of representation are hidden away from the programmer.

5 MetaML's Multi-Stage Programming Annotations

The two-level notation of Nielson & Nielson [25] features two annotations: *over-bars* to mark computations of the first stage, and *under-bars* to mark those of the second stage. Although quite powerful, this is only a subset of the annotations needed for pragmatic multi-stage programming. MetaML has four programming constructs:

- Meta-Brackets (`<_>`) are the primary means for delaying a computation. For example, whereas the expression `40+2` specifies a current (or first) stage computation, `<40+2>` specifies one for the next (or second) stage. A binary type constructor `< _ , _ >` is used to distinguish the type of the latter expression from the first one. For example, `7` has type `int`, but `<7>` has type `<int,'a>`, where (as is in ML) `'a` is a free type variable. The expression `<<(1,<2+1>)>>` has type: `<<(int * <int,'c>),'b>,'a>` and the addition will be performed in the fourth stage. The second type in the code type constructor represents the name of the context in which this code can execute. In the examples above the context is completely unconstrained hence the type variables. More about this in section 10.

- Escape (`~_`) can occur only inside enclosing meta-brackets. It is the mechanism used to insert smaller delayed computations into larger ones. Escape allows its argument to escape the "freeze" imposed by a surrounding meta-bracket and to "splice" its result into the delayed computation being built. For example:

  ```
  let val a=<1+4> in <72+~a> end
  ```

  returns the expression `<72+(1+4)>`. The escaped computation must yield a piece of code with a type that can be inserted in the context where the escape appears. The type system ensures that this is the case. For example, if `x` has type `<int,'a>`, then `<(x,1)>` has type `<(<int,'a> * int),'b>` and `<(~x,1)>` has type `<(int * int),'a>`. Objects of type code are first class citizens, and can even be λ-abstracted. For example:

  ```
  val add_72_later = fn a => <72+~a>
  ```

  declares a first class function with type `<int,'a> -> <int,'a>`, and the expression `add_72_later <8>` returns `<72+8>`. From the language designer's point of view, escape poses a very interesting technical problem, as not all uses of escape are reasonable. We discuss this issue in Section 10.1.

- Run (`run _`) takes a code-valued argument and runs it. It is the only way a computation "frozen" using meta-brackets can be computed (or "forced") in the current stage. The argument to `run` must be of *code* type. Having `run` in the language implies introducing a kind of reflection [30], and allows a future-stage computation to be performed now. To illustrate, consider the expression:

  ```
  let val a = <50-10> in 2+(run a) end
  ```

  This expression has type *int* and returns the value 42 when computed. Although run is not an annotation used in the result of BTA, it is an essential feature for a programmer who wants to use multi-stage programming to control evaluation order .

- Lift (`lift _`) allows the user to convert any *ground* value (not containing a function) into code. Contrast this with meta-brackets which converts any *syntactic expression* into a piece of code. Lift evaluates its arguments, and meta-brackets do not. Lift is most often used in conjunction with escape, because only pieces of code can be "spliced-in". For example, in the expression `<1+~(lift 2+3)>`, the escape forces `lift (2+3)` to be computed in the first stage. The addition evaluates to the value 5, and lift converts this result into the piece of code `<5>`, which is spliced (because of the escape) back into the original expression to return `<1+5>`. Lift can be used on structured values such as tuples and lists as long as they do not contain functions. For example `lift [(2,3),(2*1,4)]` evaluates to `<[(2,3),(2,4)]>`. Function values cannot be lifted using lift, as we cannot derive an intensional representation for them in general (This does not mean that function values cannot be delayed using meta-brackets. See Section 7.)

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 (`lift _`) and run (`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>`.

6 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:

```
(* 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 -> List int -> bool` [2]. A good strategy for hand annotating a program is to first

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

determine the target type of the desired annotated program. In the `member` example, the list parameter `l` is available in the first stage, and the element searched for will be available later. So, one target type for the hand-staged function is `<int,'a> -> List int -> <bool,'a>` .

Now we can begin annotating, starting with the whole expression, and working inwards until all sub-expressions are covered. At each step, we consider what annotations 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,'a> -> int list -> <bool,'a> *)
fun member v l =
   if (null l)
      then <false>
      else <if ~v=~(lift hd l)
               then true
               else ~(member v (tl l))>;
```

The annotation `~(lift hd l)` is used rather than `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 meta-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,'a>`. Moving 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,'a>` as planned.

Just like the first branch of the outer `if`-statement, the whole of the inner `if`-statement must return `bool`. So, the first branch (`true`) is fine. But because the recursive call to `member` has type `<bool,'a>`, 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>
```

The percentage sign (`%`) at the beginning of an identifier indicates that it was bound to a value in the environment in which the code was constructed. Its precise meaning will be explained in Sections 7 and 10.2.

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 were: first, it should be easy to write 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 where different from those of other multi-stage systems. In particular, we consider the following quality crucial to MetaML:

Cross-stage Persistence: *A variable i bound in stage n, will be available in stages n, $n+1$ and all future stages.*

To the user, this means the ability to stage non-closed expressions. Non-closed expressions, like λ-abstractions with free variables, must resolve their free variable occurrences in the static environment where the meta-bracketed expression occurs. One can think of a code value as containing an environment which binds its free variables. For example the expression,

```
let val a=1+4 in <72+a> end
```

returns a value `<72+%a>`. The % sign indicates that the free variable `a` is bound in the value's local environment. The % sign is printed by the display mechanism. The variable `a` has been bound during the first stage to the constant 5. In fact, in MetaML `%a` is not a variable, but rather, a new *constant*, and the name "a" is only hint to the user about where this constant originated. When `%a` is evaluated in a later stage, it will return 5 independent 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 delayed using this hygienic binding mechanism.

Specifying this behavior turns out to be non-trivial. In an interpreter for a multi-stage language, this requirement manifests itself as complex variable-binding rules, the use of closures, or capture-free substitutions. Our semantics addresses this in a rather unique way (See Section 10.2).

Cross-Stage Persistence poses a problem when staging is used for program generation. If the first stage is performed on one computer, and the second on another, we must "port" the local environments from the first machine to the second. Since arbitrary objects, such as functions and closures, can be bound in these local environments this can become a problem. Currently, MetaML assumes that the computing environment does not change between stages. This is part of what we mean by having an integrated system.

Cross-Stage Persistence can be relaxed by allowing variables to be available at exactly one stage. This seems to have been the case in all multi-stage languages known to us to date [25, 12, 9, 10, 7, 6]. The primary difficulty in implementing persistence is the proper hygienic treatment of free variables. We will show how this problem can be solved, thus allowing the user to stage significantly more expressions than was previously possible.

But even in MetaML, it will not be possible to stage every expression in the language. In particular, we must ensure that the user can only specify computations that respect the following condition:

Cross-Stage Safety: *An input first available at stage m cannot be used at a stage n if $m > n$.*

The problem arises with the use of the escape annotation. In particular, consider the expression

```
fn a => <fn b => ~(a+b)>
```

which is an (incorrectly) staged version of the function $\lambda a.\lambda b.a+b$. Operationally, the annotations require 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" [22] in statically-typed languages. A formal proof of this property is ongoing work.

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

8 Isomorphism for Code Types

Recall the types of the staged `member` function: `<int,'a> -> List int -> <bool,'a>`, and the type of the term `<fn x => (member <x> [1,2,3])>` which is: `<int -> bool,'a>` . This suggests that a function from code to code can be turned into the code of a function. This is important to users because `<alpha,'a> -> <beta,'a>` is a function and cannot be printed or observed, while `<alpha -> beta,'a>` is a representation of a function, and can be printed and observed. We can define two functions to convert between these two types:

```
(*  back: <'A,'c> -> <'B,'c>  -> <('A -> 'B),'c> *)
fun back f = <fn x => ~(f <x>)>;

(*  forth: <('A -> 'B),'c> -> (<'A,'c> -> <'B,'c>) *)
fun forth f x = <~f ~x>;
```

Here we use capitalized type variables to distinguish the type in the code from the context the code must evaluate in.

The conversion is not between syntactic forms, but semantic values. For example, the code produced by an application of `back` is in a language extended with a new construct that allows us to embed any value into syntax, without needing to know about its intentional representation. Thus, we are really not converting functions into source code, but rather, returning syntax that denotes this function under our semantics. Under this proviso (and disregarding termination issues) the composition of these two functions is identity under MetaML's semantics (see Section 10). They define an isomorphism between values of type `<A,'c> -> <B,'c>` and `<A -> B,'c>`. [3].

The `back` and `forth` functions 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.

This isomorphism can also be viewed as a formalization of the intuitive equivalence of a symbolic evaluator [23] `<A,'c> -> <B,'c>` and the syntactic representation of a function `<A -> B,'c>`. It seems that this isomorphism, which MetaML has allowed us to make concrete, is at the heart of concise reduction systems, such as Danvy's type-directed partial evaluator [4] and its extensions [27]. Under MetaML's semantics, we can switch between the two types without needing to worry about substitution or variable capture.

This has profound implications for the writing of staged functions. In our experience annotating a function to have type `<A,'c> -> <B,'c>` requires less annotations than annotating it to have type `<A -> B,'c>` 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 inspect the code.

The type of `back` is one of the axioms of the logic system motivating the type system of Davies [6]. MetaML's type system was motivated purely by operational reasons. At the same time, it is important for the programmer to have both coercions, thereby being able to switch back and forth between the two isomorphic types as the need arises.

This becomes even more important when writing programs with more than two stages. Consider the function:

```
fun back2 f = <fn x => <fn y => ~~(f <x> <<y>>)>>;
back2 : (<A,'d> -> <<B,'e>,'d> -> <<C,'e>,'d>)
              -> <A -> <B -> C,'e>,'d>
```

This allows us to write a program which takes a `<a,'d>` and a `<<b,'e>,'d>` as arguments and which produces a `<<c,'e>,'d>` and stage it into a three-stage function. Our experience is that such functions have considerably fewer annotations, and are easier to think about. We illustrate this in the next section.

9 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 for each column of the second. This example has appeared in several other works [9, 20] and we give our version below. 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 `<` and `>` because of the possible confusion with meta-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 inference mechanism was 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 "see" what the results are immediately.

```
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 inlined the calls to add it would be exponential. This is why being able to have free variables (such as add) in code is indispensable. Now let us observe the result of the second stage computation:

```
val f4 = (eval 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 are evident in the third stage program. If the vector is sparse then this sort of optimization can have dramatic effects.

10 Semantics of $\lambda^{\mathcal{M}}$

Figure 1 presents the static and dynamic semantics of the meta-lambda calculus, $\lambda^{\mathcal{M}}$. This calculus is a mini-MetaML, which illustrates the relevant features of the staging annotations on the semantics of MetaML.

$\lambda^{\mathcal{M}}$ is a call by value lambda calculus which supports integers, functions, and code $(\texttt{int} \mid t \rightarrow t \mid \langle t \rangle^{t} \mid \alpha)$. The syntax of terms includes integer constants, variables, applications, abstractions $(i \mid x \mid e\ e \mid \lambda x^{t} . e)$ and the four staging annotations: meta brackets, escape and run $(\texttt{<}e\texttt{>} \mid \texttt{\~{}}e \mid \texttt{run}\ e)$. In addition, the constant operator $(\uparrow v)$ allows us to injects a value into a term, and is crucial to the conciseness of our implementation of Cross-Stage Persistence. It is these constants that we print out as a % followed by a name. Note that users do not write programs with the constant operator; it is only introduced during reduction. Every shift in stage from a lower stage to a higher stage enriches the syntax passed to the higher stage with a new set of constants; the values of the previous stage that could still be referenced in the future.

Finally, we note that the lift annotation of MetaML is does not appear explicitly in the semantics of $\lambda^{\mathcal{M}}$. Instead, the lift can be defined within $\lambda^{\mathcal{M}}$for each type.

10.1 Static Semantics

The static semantics is expressed as a set of inference rules that determine if a term is well-formed, and determine its type. The judgement $\Sigma\ \Delta \overset{n}{\vdash} x : \tau_1, \tau_2$ is read *under the context stack Σ, the type environment Δ, the term x has type τ_1 at level n and may execute in the context with* name τ_2.

The intuition behind contexts, is that any expression can only execute in a context which contains bindings for its free variables. The type inference algorithm assigns the same context name to expressions that must execute in the same context.

The type assignment Δ maps variable to types and levels and context names. Every variable is bound at some particular level, namely, the level of the abstraction in which it is bound (**Abs** rule). The role of n in the judgement $\Sigma\,\Delta \overset{n}{\vdash} x : \sigma$ is to keep track of the level of the expression being typed. 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 meta-brackets (**Bracket**), and decremented for one inside an escape (**Escape**). Note that the rule **Escape** does not allow escape to appear at level 0. In other words, escape must appear inside uncancelled meta-brackets.

There are three main kinds of errors related to staging annotations that can occur at runtime:

- A variable is used in a stage before it is available, or
- Run or escape are passed values having a non-*code* type, or
- Run is passed a code-type value with free variables. This manifests itself in the type, where the name of the context is constrained.

The first kind of error is checked by the **Var** rules. Having no rule for $m > n$ enforces Cross-Stage Persistence: Variables available in the current stage (m) 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 an important contribution of this paper, and is accomplished by the free variable check in the rule **Run n**: *Only code whose context is completely unconstrained my be run.*

For the standard part of the language, *code* (now denoted by $\langle\_\rangle$ for conciseness) is a normal type constructor that needs no special treatment and the level n is never changed. Similar type systems have been identified and used by Gomard and Jones [11], Davies & Pfenning [6] and Davies [7].

An important difference between these type systems and the one in Figure 1 is that in all previous statically-typed multi-stage languages [25, 7, 6], only the following *monolithic* type rule is used for variables:

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

Whereas we allow the more general condition $m \leq n$. This means any generated expressions may as well be evaluated in the empty environment since all well-typed terms are closed terms and cannot reference any free variables. For example the expression:

```
val lift_like = fn x => <x>
```

is accepted, because inside the meta-brackets, $n = 1$, and $(\Delta\ \mathtt{x}) = \alpha^0$. 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 `<%x>`, and not a literal expression denoting the value. But this example demonstrates that meta-brackets can be used to "lift" any value, including functions. This is explained in the dynamic semantics.

The type system rejects the expression

```
fn a => <fn b => ~(a+b)>
```

because, inside the escape, $n = 0$, and $(\Gamma\ \mathtt{b}) = \alpha^1$, but $1 > 0$.

10.2 Dynamic Semantics

The dynamic semantics provides meaning to terms. Values are a subset of terms, and we denote them with a small diamond superscript ($i^\diamond \mid \{\lambda x^t.e\}^\diamond \mid \langle e\rangle^\diamond$). The semantics given in Figure 1, when applied to well-typed terms, maintains the invariant that no free variables ever occur in code values which will later be `run`.

The dynamic semantics is broken into two sets of rules, reduction and rebuilding. Reduction ($\Gamma \vdash e \hookrightarrow v$) maps terms to values and rebuilding ($\Gamma \vdash e \overset{n+1}{\hookrightarrow} e$) maps terms to terms and is indexed by a level $n+1$. Rebuilding "reconstructs" terms under the environment Γ.

The environment Γ binds a variable to a value. Bindings in environments come in two flavors: real ($\mathrm{Real}(v)$) and symbolic ($\mathrm{Sym}(x)^t$). The extension of the environment with real values occurs only in the rule **App 0**. Such values are returned under reduction (**Var 0**), or injected into constant terms (**RVar n+1**) under rebuilding.

Several things about rebuilding should be noted:

1. Rebuilding replaces all known variables with a constant expression ($\uparrow v$) where the v comes from $\mathrm{Real}(v)$ bindings in Γ (**RVar n+1**).
2. Rebuilding renames all bound variables. Symbolic $\mathrm{Sym}(x')^t$ bindings occur in rules **Abs 0** and **Abs n+1** where a term is rebuilt, and new names must be introduced to avoid potential variable capture. These new names are projected from the environment in rule **SVar n+1**.
3. Rebuilding executes escaped expressions to obtain code to "splice" into the context where the escaped term occurs (**Escape 1**).

Without the staging annotations, rebuilding is simply capture-free substitution of the symbolic variables bound in Γ. Rebuilding is initiated in two places, in rule **Abs 0** where it is used for capture-free substitution, and in rule **Bracket 0** where it is applied to terms inside dynamic brackets and it describes how the delayed computations inside a dynamic value are constructed.

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 rebuilding actually implements capture-free substitution as advertised.

The rules **Escape 1**, **Run 0**, and **Bracket 0** are at the heart of the dynamic semantics.

In the rebuilding rule **Escape 1**, an escaped expression at level 1 indicates a computation must produce a code valued result ($\langle e_2\rangle^\diamond$), and rebuilding returns the term e_2.

The reduction rule **Bracket 0** describes how a code value is constructed from a meta-bracketed term $\mathtt{<}e_1\mathtt{>}$. The embedded expression is rebuilt at level 1, and the returned term is injected into the domain of values.

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 all free variables in the original code valued term have been replaced by constant expressions ($\uparrow v$).

Domains and Relations

levels	n	$\to$	$0 \mid 1 \mid n+1 \mid n+2 \mid \ldots$		
integers	i	$\to$	$\ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$		
types	τ	$\to$	$\mathtt{int} \mid \tau \to \tau \mid \langle \tau \rangle^{\tau} \mid \alpha$		
terms	e	$\to$	$i \mid x \mid e\, e \mid \lambda x^{\tau}.e \mid \texttt{<}e\texttt{>} \mid \texttt{\~{}}e \mid \mathbf{run}\, e \mid \uparrow v$		
values	v	$\to$	$i^{\diamond} \mid \{\lambda x^{t}.e\}^{\diamond} \mid \langle e \rangle^{\diamond}$		
bindings	b	$\to$	$\mathrm{Real}(v) \mid \mathrm{Sym}(x)^{t}$		
environments	Γ	$\to$	$\bullet \mid \Gamma, x \mapsto b$	**where** $(\Gamma, x \mapsto b)y$	$\equiv$ if $x = y$ then b else $\Gamma\, y$
type environments	Δ	$\to$	$\circ \mid \Delta, x \mapsto (\tau, \alpha)^{n}$	**where** $(\Delta, x \mapsto (\tau,\alpha)^{n})y$	$\equiv$ if $x = y$ then $(\tau,\alpha)^{n}$ else $\Delta\, y$
context stacks	Σ	$\to$	$[\,] \mid \alpha;\Sigma$		

reduction $\Gamma \vdash e \hookrightarrow v$ rebuilding *at level* n $\Gamma \vdash e \overset{n}{\hookrightarrow} e$ term typing *at level* n $\Sigma\,\Delta \overset{n}{\vdash} e : \tau, \tau$

Static Semantics

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

Var =0: $\dfrac{\Delta\, x = (\tau, \alpha_1)^{0}}{\Sigma\,\Delta \overset{n}{\vdash} x : \tau, \alpha_2}$

Var ≤n: $\dfrac{\Delta\, x = (\tau, \alpha)^{i} \quad i \neq 0 \wedge i \leq n}{\Sigma\,\Delta \overset{n}{\vdash} x : \tau, \alpha}$

Br n: $\dfrac{(\alpha_2;\Sigma)\,\Delta \overset{n+1}{\vdash} e : \tau, \alpha_1}{\Sigma\,\Delta \overset{n}{\vdash} \texttt{<}e\texttt{>} : \langle \tau \rangle^{\alpha_1}, \alpha_2}$

Abs n: $\dfrac{\Sigma\,\Delta, x \mapsto (\tau_1, \alpha)^{n} \overset{n}{\vdash} e : \tau_2, \alpha}{\Sigma\,\Delta \overset{n}{\vdash} \lambda x^{\tau_1}.e : \tau_1 \to \tau_2, \alpha}$

Esc n+1: $\dfrac{\Sigma\,\Delta \overset{n}{\vdash} e : \langle \tau \rangle^{\alpha_1}, \alpha_2}{(\alpha_2;\Sigma)\,\Delta \overset{n+1}{\vdash} \texttt{\~{}}e : \tau, \alpha_1}$

Run n: $\dfrac{\alpha_2 \notin FV(\Delta, \tau) \qquad \Sigma\,\Delta \overset{n}{\vdash} e : \langle \tau \rangle^{\alpha_2}, \alpha_1}{\Sigma\,\Delta \overset{n}{\vdash} \mathbf{run}\, e : \tau, \alpha_1}$

App n: $\dfrac{\Sigma\,\Delta \overset{n}{\vdash} e_1 : \tau_1 \to \tau, \alpha \qquad \Sigma\,\Delta \overset{n}{\vdash} e_2 : \tau_1, \alpha}{\Sigma\,\Delta \overset{n}{\vdash} e_1\, e_2 : \tau, \alpha}$

Con n:: $\dfrac{\star\text{Never appears in source terms}}{\Sigma\,\Delta \overset{n}{\vdash} \uparrow v : \star}$

Dynamic Semantics

Int 0: $\Gamma \vdash i \hookrightarrow i^{\diamond}$

Int n+1: $\Gamma \vdash i \overset{n+1}{\hookrightarrow} i$

Abs 0: $\dfrac{\Gamma, x \mapsto \mathrm{Sym}(x')^{\tau} \vdash e \overset{1}{\hookrightarrow} e_1}{\Gamma \vdash \lambda x^{\tau}.e \hookrightarrow \{\lambda x'^{\tau}.e_1\}^{\diamond}}$

Abs n+1: $\dfrac{\Gamma, x \mapsto \mathrm{Sym}(x')^{\tau} \vdash e_1 \overset{n+1}{\hookrightarrow} e_2}{\Gamma \vdash \lambda x^{\tau}.e_1 \overset{n+1}{\hookrightarrow} \lambda x'^{\tau}.e_2}$

App 0: $\dfrac{\Gamma \vdash e_1 \hookrightarrow \{\lambda x^{\tau}.e\}^{\diamond} \qquad \Gamma \vdash e_2 \hookrightarrow v_2 \qquad \bullet, x \mapsto \mathrm{Real}(v_2) \vdash e \hookrightarrow v}{\Gamma \vdash e_1\, e_2 \hookrightarrow v}$

App n+1: $\dfrac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} e_3 \qquad \Gamma \vdash e_2 \overset{n+1}{\hookrightarrow} e_4}{\Gamma \vdash e_1\, e_2 \overset{n+1}{\hookrightarrow} e_3\, e_4}$

Var 0: $\dfrac{\Gamma\, x = \mathrm{Real}(v)}{\Gamma \vdash x \hookrightarrow v}$

SVar n+1: $\dfrac{\Gamma\, x = \mathrm{Sym}(x')^{\tau}}{\Gamma \vdash x \overset{n+1}{\hookrightarrow} x'}$

RVar n+1: $\dfrac{\Gamma\, x = \mathrm{Real}(v)}{\Gamma \vdash x \overset{n+1}{\hookrightarrow} \uparrow v}$

EVar n+1: $\dfrac{x \notin \Gamma}{\Gamma \vdash x \overset{n+1}{\hookrightarrow} x}$

Bracket 0: $\dfrac{\Gamma \vdash e_1 \overset{1}{\hookrightarrow} e_2}{\Gamma \vdash \texttt{<}e_1\texttt{>} \hookrightarrow \langle e_2 \rangle^{\diamond}}$

Bracket n+1: $\dfrac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} e_2}{\Gamma \vdash \texttt{<}e_1\texttt{>} \overset{n}{\hookrightarrow} \texttt{<}e_2\texttt{>}}$

Escape 1: $\dfrac{\Gamma \vdash e_1 \hookrightarrow \langle e_2 \rangle^{\diamond}}{\Gamma \vdash \texttt{\~{}}e_1 \overset{1}{\hookrightarrow} e_2}$

Escape n+2: $\dfrac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} e_2}{\Gamma \vdash \texttt{\~{}}e_1 \overset{n+2}{\hookrightarrow} \texttt{\~{}}e_2}$

Run 0: $\dfrac{\Gamma \vdash e \hookrightarrow \langle e_1 \rangle^{\diamond} \qquad \bullet \vdash e_1 \hookrightarrow v_1}{\Gamma \vdash \mathbf{run}\, e \hookrightarrow v_1}$

Run n+1: $\dfrac{\Gamma \vdash e_1 \overset{n+1}{\hookrightarrow} e_2}{\Gamma \vdash \mathbf{run}\, e_1 \overset{n+1}{\hookrightarrow} \mathbf{run}\, e_2}$

Constant 0: $\Gamma \vdash \uparrow v \hookrightarrow v$

Constant n+1: $\Gamma \vdash \uparrow v \overset{n+1}{\hookrightarrow} \uparrow v$

Figure 1: **The Semantics of $\lambda^{\mathcal{M}}$**

11 Optimizations

11.1 Safe Beta Reduction

To write multi-stage programs effectively, one needs to observe the programs produced, and these programs should be as simple as possible. For this reason, our implementation performs automatic *safe-beta reduction* on constants and variables. 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 [26]. 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>`.

We realize of course that this might make it hard to understand *why* a particular program was generated. In our experience, the resulting smaller, simpler programs, are easier to understand and make this tradeoff worthwhile.

11.2 Nested Escapes

When we first wrote programs with more than two levels we observed that our programs took a long time to run. We traced this to rule **Escape n+2** of our semantics. 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`.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{e \hookrightarrow \langle \texttt{<5>} \rangle^{\diamond}}{\sim e \overset{1}{\hookrightarrow} \texttt{<5>}}}{\sim\sim e \overset{2}{\hookrightarrow} \sim\texttt{<5>}}}{\texttt{<}\sim\sim e\texttt{>} \overset{1}{\hookrightarrow} \texttt{<}\sim\texttt{<5>>}}}{\texttt{<<}\sim\sim e\texttt{>>} \hookrightarrow \langle \texttt{<}\sim\texttt{<5>>} \rangle^{\diamond}} \qquad \dfrac{\dfrac{\dfrac{5 \overset{1}{\hookrightarrow} 5}{\texttt{<5>} \hookrightarrow \langle 5\rangle^{\diamond}}}{\sim\texttt{<5>} \overset{1}{\hookrightarrow} 5}}{\texttt{<}\sim\texttt{<5>>} \hookrightarrow \langle 5 \rangle^{\diamond}}}{\texttt{run <<}\sim\sim e\texttt{>>} \hookrightarrow \langle 5 \rangle^{\diamond}}$$

For two levels the term is rebuilt 2 times. For three levels the term is rebuilt 3 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**.

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

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

Thus a long sequence of escapes surrounded by an equal number of brackets gets rebuilt exactly one. This optimization is safe since there are no variables in a rebuilt term. So rebuilding it more than once performs no useful work. This correctness of this optimization follows from the fact that under our semantics inside nesting brackets $\sim$`<`e`>` is always equal to e.

12 Discussion and Related Works

A summary of the distinguishing characteristics of other work on multi-stage languages is shown in figure 2. In the figure we list whether or not that work contains the properties we believe to be important. Those properties are:

- Staging: Staging can be expressed in all these languages, in the sense that code representing expressions can be expressed, and that applications can be delayed to future stages. In this row, "2" means two stages are supported, and "+" means arbitrary number of stages is supported.
- Strong Typing: In a single stage language, strong typing ensures that a well-typed program "cannot go wrong" by applying functions to arguments of the wrong type. In addition, multi-stage programs can go wrong if a computation attempts to use a variable before it is available. A strongly typed multi-stage language protects against both kinds of errors.
- Monolithic Variables: Whether a variable from one stage can be used in all reasonable ways at that same stage. In the example, `f` is applied to `<x>`, and hence `f` is acting as a "code transformer".
- `run` or *eval* function: The languages also come with different set of multi-stage programming constructs. In particular, not all the language allow reflection in the form of a `run` or *eval* function. Most notably, even in the most recent work of Davies, it was not known how *eval* could be included in the language [6].
- Cross-Stage Persistence: This is the most distinguishing feature of MetaML, and has been discussed in detail in this paper. To our knowledge, this feature has not been proposed or incorporated into any multi-stage programming language. In essence, cross-stage persistence means that the programmer can use a variable bound in any stage in an expression executed in any future stage. Alternatively, it allows us to stage expressions containing free variables, as long as the type system is satisfied that these variables are available before this expression is evaluated.
- Cross-Platform Portability: The ability to print generated code as text. This allows code generated on one machine to be compiled and run on other machines. If code embeds its own "local environment" this becomes considerably more difficult. The loss of this ability is the price paid for cross stage persistence.

In what follows is an historical perspective of the work highlighted in the table:

- NN: Nielson and Nielson pioneered the investigation of staged languages with their two-level functional language [25, 24]. They presented rules for the well-formedness of the binding-times of expressions in the

Facility	**Example**	NN [25]	GA [11]	GB [9]	Th [31]	HG [12]	$\lambda^{\Box}$ [7]	$\lambda^{\bigcirc}$ [6]	$\lambda^{\mathcal{M}}$
Staging	`<λx.x>`	2	2	+	2	+	+	+	+
Strong Typing		Y	1	N	N	N	Y	Y	Y
Monolithic Variables	`<λx.~(f <x>)>`	Y	Y	Y	Y	Y	N	Y	Y
Reflection	`run` or *eval*	N	N	N	Y	N	Y	N	Y
X-Stage Persistance	`λf.<λx.f x>`	N	N	N	N	N	N	N	Y
X-Platform Portability		Y	Y	Y	Y	Y	Y	Y	N

Figure 2: Comparative feature set

language, from which MetaML's type rules are derived. They also sketched guidelines for a multi-stage ("*B*-level") language. The two-level language is widely used to describe binding-time annotations in the partial evaluation literature.

- GA: Gomard and Jones use a statically-typed two-stage language for partial evaluation of the untyped λ-calculus [11]. 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 formal semantics is inspired by their work.

- GB: Glück and Jørgensen [9] present the novel idea of multi-level BTA, as a very efficient and effective alternate to multiple self-application. An untyped multi-level language based on Scheme is used for the presentation. Our study of MetaML is at a more basic level: MetaML is an abstract calculus. It is also notable that all intermediate results in GJ are printable, i.e., have an 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 makes the proper specification of MetaML more difficult. For example, we can't use [9]'s "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 constraints-solving methods to perform the analysis. The MBTA is type-based, but underlying language is dynamically typed.

- Th: Thiemann [31] studies a two-level language with `eval`, `apply`, and `call/cc`, in the context of studying the partial evaluation of a greater subset of scheme than was done previously. A BTA based on constraint-solving is presented. Although the problems with `eval` and `call/cc` are highlighted, and unlike with MetaML, there is no explicit notion of partially static types, and so, the complexity of introducing `eval` into a multi-stage language does not manifest itself. Thiemann also deals with the issue of variable-arity functions, which is a practical problem when dealing with `eval` in Scheme.

- HG: Hatcliff & 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 [12]. 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 & Glück identified *language-independence* of the internal representation of "code" as an important characteristic of any multi-stage language.

- $\lambda^{\Box}$: Davies & Pfenning presented the first statically-typed multi-stage language Mini-ML$^{\Box}$ [6]. The type system is motivated by constructive modal logic, and a formal proof is presented for the equivalence of binding-time correctness and modal correctness. In contrast, the MetaML type-system was motivated primarily by operational considerations. Despite the different origins, the languages have a lot in common: meta-brackets, escape, and run roughly correspond to box, unbox, and eval respectively. Mini-ML$^{\Box}$'s $\Box$ type constructor is also similar to *code*. Interestingly, in Mini-ML$^{\Box}$, eval is defined in terms of unbox, whereas in MetaML, neither run or escape can be defined in terms of each other. Also, while Mini-ML$^{\Box}$ can simulate persistence for code values, a stage-zero function, for example, cannot be made persistent. Finally, Mini-ML$^{\Box}$ allows expressing functions like `back` are not expressible in the language.

- $\lambda^{\bigcirc}$: 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 meta-brackets and escape. The type constructor $\bigcirc$ also corresponds (roughly) to *code*. Unfortunately, `eval` is no longer expressible in the language.

Sheard has also investigated richer type systems for multi-staged programming. Sheard and Nelson investigated a two-stage language for the purpose of program generation [28]. 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 [29] investigate a dynamic type systems for multi-staged programs where some type obligations of staged computations can be put off till run-time.

The type rule for `run` presented in this paper is motivated by the type system for `runST` [19].

13 Conclusion

We have described an n-stage 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 find the following features 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, i.e. macros which bind identifiers in the environment of definition, which are "captured" in the environment of use.
- **Multi-stage aware type system.** The type checker reports phase errors as well as type errors is crucial when debugging multi-stage programs, thus ensuring Cross-Stage Safety.
- **Display of code.** When debugging, it is important for users to observe the code produced by their programs. This implies a display mechanism (pretty-printer) for values of type code.
- **Display of Constants.** Constants originating from persistent variables are hard to identify. The % tags provide an approximation of where these constants came from. While potentially misleading they are often quite useful.
- **The Isomorphism between `<A,'c> -> <B,'c>` and `<A -> B,'c>`.** The isomorphism (which can only be expressed because of cross-stage persistence), reduces, sometimes drastically, the number of annotations needed to stage multi-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 never necessary (because of cross-stage persistence) it helps produce code which is easier to understand, because constants become explicit.
- **Safe beta reduction.** Safe beta reduction of the application of explicit abstractions to explicit variables reduces the clutter in generated code.

To further illustrate this we provide an extended example in Appendix A which stages a term rewriting system in which the the rewriting rules become known in the first stage and the terms to be rewritten become known only in later stages.

14 Future Work

We have built an implementation which was used to program the examples in this paper. Currently, the implementation supports polymorphic type-inference so that type information on bound variables is not necessary. We are currently extending this implementation to include all the features in core-ML.

We are also actively pursuing a subject reduction theorem for $\lambda^{\mathcal{M}}$. The multi-level syntax makes the syntactic approaches to type soundness [32] difficult to apply, because reduction contexts may appear inside lambda expressions at levels greater than zero. We have also found that the non-Hindley-Milner type judgement for the run annotation complicates matters considerably.

References

[1] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. New York: ACM, Berlin: Springer-Verlag, 1991.

[2] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, January 1993.

[3] Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design.* Progress in Theoretical Computer Science. Birkhäuser, 1995.

[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, 11$^{\text{th}}$ 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] Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 65:122–157, 1985.

[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] 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.

[13] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Cobminators and λ-Calculus.* Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986.

[14] 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.

[15] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[16] 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.

[17] 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.

[18] 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.

[19] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *Proceedings of the International Conference on Functional Programming*, Amsterdam, June 1997.

[20] 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.

[21] 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.

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[23] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Functional Programming*, 2(3):345–364, July 1992.

[24] F. 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.

[25] F. Nielson and H. R. Nielson. *Two-Level Functional Languages.* Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

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

[27] 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.

[28] 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.

[29] 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. (to appear).

[30] Brian C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, January 1982.

[31] Peter Thiemann. Towards partial evaluation of full Scheme. In Gregor Kiczales, editor, *Reflection 96*, pages 95–106, San Francisco, CA, USA, April 1996.

[32] Wright and Felleisen. A syntactic approach to type soundness. *Information and Computation (formerly Information and Control)*, 115, 1994.

A Term Rewriting: An Extended Example

In this section we provide an extended example which illustrates multi-stage programming.

Dershowitz [8] defines a *term-rewriting system* as a set of directed rules. Each rule is made of a left-hand side and a right-hand side. A rule may be applied to a term t if a subterm s of t matches the left-hand under some substitution σ. A rule is applied by replacing s with t', where t' is the result of applying the substitution σ to the right-hand side. We say *"t rewrites (in one step) to t′"*, and write $t \Rightarrow t'$. The choice of which rule to apply is made non-deterministically. As an example, here are the rules for a Monoid [8]:

$$\begin{array}{lrcl} r_1: & x+0 & \rightarrow & x \\ r_2: & 0+x & \rightarrow & x \\ r_3: & x+(y+z) & \rightarrow & (x+y)+z \end{array}$$

Variables x, y, and z in the rules can each match any term. If a variable occurs more than once on the left-hand side of a rule, all occurrences must match identical terms. We call this the *compatibility condition.* These rules allow us to have derivations such as:

$$\begin{array}{ll} & (a+b)+\underline{(0+(d+e))} \\ \Rightarrow & \text{by } r_2, \sigma=[(d+e)/x] \\ & \underline{(a+b)+(d+e)} \\ \Rightarrow & \text{by } r_3, \sigma=[(a+b)/x, d/y, e/z] \\ & ((a+b)+d)+e \end{array}$$

where the subterm being rewritten (s) has been underlined.

Generally, the rules do not change over the life of the system. At the same time, the basic form of the matching function is a simultaneous traversal of a subject term and the left-hand side of the rule it is being matched against.

This offers an opportunity for staging: We can "specialize" matching over the rules in a first stage, and eliminate the overhead of traversing the left-hand side of the rules. Not only that, but as we will see, we can also remove a significant amount of administrative computations involved in constructing and applying the substitution σ. One would expect that this would significantly speed up the rewriting system.

Terms can be implemented using the following MetaML type:

```
datatype 'a T = Const of int
              | Var of string
              | Op of ('a * string * 'a);
```

The declaration introduces a type constructor `T` parameterized by another type, `'a`. The constructors of this type are `Const`, `Var` and `Op` representing one level of an integer, variable, or infix binary term, respectively. The following definitions will also be used:

```
datatype Term = In of (term T);
datatype 'a Maybe = Nothing of unit | Just of 'a;
type Sub = ((string * term) List) Maybe;
type Rule = (Term * Term);

fun const x = In (Const x);
fun var x = In (Var x);
fun op x  = In (Op x);
fun out (In x) = x;
```

where `Term` ties the recursive knot in `T` to represent terms, and `Sub` is the type of substitutions, respectively. The `Maybe` type provides a mechanism to write functions that can fail to return a meaningful value by returning `Nothing ()` instead. Since pattern matching may fail the substitution type is a `Maybe` type. We represent rules as an ordered pair of terms. Thus, r_1 would be `(op(var "x","+",int "0"),var "x")`. The function `out : Term -> Term T` removes a term from the recursive type to the 1 level type.

To simplify, we focus on `match`, a function that matches a rule with the whole subject term, that is, the special case when $s = t$. We assume that a separate helper function applies `match` to all subterms of the subject term.

Now, let us consider what the result of specializing `match` over a rule should look like. If we take $r_1 : x + 0 \to x$ as an example, then an good result should have a form close to:

```
fun rewriteR1 term =
case (out term) of
  Op (t1,s,t2) =>
      if streq(s,"+")
      then (case (out t2) of
              Const n => if n=0 then t1 else term
            | _ => term)
      else term
| _ => term;
```

Here there is no interpretive overhead of traversing the rule, and the substitution operation has also been computed in the first stage. In the rest of this section, we will see how far towards this goal we can go using MetaML.

A.1 An Implementation of `match`

Figure 3 presents the complete code for `match`, together with a staged version. Note that if we erase the annotations from the staged version, we get the source version back. The source `match` function takes a pattern term `pat`, a substitution `msigma`, a term `term`, and returns a substitution. Recall that a substitution is a maybe type and may be `Nothing()`.

Hence, the type of the `match` function is $\Theta \to \Sigma \to \Theta \to \Sigma$, where $\Theta \equiv$ `Term`, and $\Sigma \equiv$ `Sub`.

If `match` is passed an invalid substitution `Nothing ()` the outer `case`-statement propagates the failure. Otherwise, if the pattern term is a variable, the substitution is extended appropriately after checking the compatibility condition. Similarly, if the pattern term is an operator, it must be checked that the subject term is also an operator with the same operation, and then the right- and left-hand-sides of the pattern and the subject term are recursively matched, extending the substitution. If the pattern term is an integer, then the subject term must also be an integer with the same value. In this case the original substitution is returned. In all other cases, `match` returns `Nothing ()`, indicating that `match` has failed.

The staged `match` function has type $\Theta \to< \Sigma, \texttt{'a} >\to$ [4] $< \Theta, \texttt{'a} >\to< \Sigma, \texttt{'a} >$. The type indicates that the pattern term is inspected only in the first stage, and the result is a specialized function that can be run in a future stage. We can also define and annotate `rewrite` using the helper function `wrapper` as follows:

```
(*  wrapper : Rule -> <Term> -> <Term>  *)
fun wrapper (lhs,rhs) term =
let val ms  = match lhs <Just []> term
in
  < case (~ms) of
      Nothing () => ~term
    | Just (sigma) =>
         subst sigma rhs > end ;

(*  rewrite : Rule -> <Term -> Term>  *)
fun rewrite rule = <fn x => ~(wrapper rule <x>)>;
```

In Figure 4 the code generated for $r_1 : x+0 \to x$ appears. The traversal of the pattern term has been performed. Yet, compared with the code we derived by hand, there are too many nested `case`-statements, and the calls to `subst` and `compatible` have not been performed.

Careful inspection shows that it should, in fact, be possible to reduce the nested `case`-statements by meaning-preserving transformations. If the outer `case`-statements could be "pushed" through the inner ones, then we would be able to simplify all the values at the leaves of the inner `if`-statements. In particular, in every case where we return `Nothing ()`, the unchanged term `t1` would be returned, and where `Just` (σ) is returned then the substitution `subst` σ could be performed on `rhs`.

Given that we were able to write the function `rewriteR1` by hand, it should be clear that values of `Maybe`-type need not appear in the generated code. Unfortunately, this cannot be achieved by using only the staging annotations on the current program.

In particular, the test of the compatibility condition in `match` cannot be performed at generation time since the subject term is unknown. At the same time, this call determines whether `Nothing` or `Just` is returned. This implies `match` must return a value of type $< \texttt{Maybe}\alpha, \texttt{'a} >$. This means the generated code will contain values of `Maybe` type.

Similarly, threading `msigma` through the recursive calls to `match` stops us from being able to reduce the calls to `compatible` and `subst` at generation time: The result of both of these functions depends on the substitution list,

```
(* Source match: Term -> Sub -> Term -> Sub *)

fun match pat msigma term =
case (msigma) of
   Nothing () => Nothing ()
 | Just (sigma) =>
(case (out pat) of
    Var u =>
      if compatible u sigma term
         then Just (cons((u,term),sigma))
        else Nothing ()
  | Op (t11,s1,t12) =>
      (case (out term) of
          Op (t21,s2,t22) =>
        (if streq(s2,( s1))
            then (match t11
                    (match t12 msigma t22)
                       t21)
            else Nothing ())
        | _ => Nothing ())
  | Const n =>
      (case (out term) of
         Const u => if u=n
                     then msigma
                     else Nothing ()
       | _ => Nothing ()));
```

```
(* Annotated match: Term -> <Sub> -> <Term> -> <Sub> *)

fun match pat msigma term =
<case ~msigma of
    Nothing () => Nothing ()
  | Just (sigma) =>
 ~(case (out pat) of
      Var u =>
        <if compatible u sigma ~term
            then Just (cons((u,~term),sigma))
            else Nothing ()>
    | Op (t11,s1,t12) =>
        <case (out ~term) of
            Op (t21,s2,t22) =>
          (if streq(s2, ~(lift s1))
              then ~(match t11
                         (match t12 msigma <t22>)
                            <t21>)
              else Nothing ())
          | _ => Nothing ()>
    | Const n =>
        <case (out ~term) of
           Const u => if u= ~(lift n)
                         then ~msigma
                         else Nothing ()
         | _ => Nothing ()>)>;
```

Figure 3: Normal and Annotated versions of the function `match`.

```
<(fn t1 =>
     (case (case %out t1 of
              Op(tl,op1,tr) =>
                 if %streq (op1,"+")
                    then (case (case %out tr of
                                   Const n =>
                                      if %= (n,0) then Just(nil) else Nothing ()
                                  | Var s => Nothing ()
                                  | Op s => Nothing ()) of
                             Nothing () => Nothing ()
                            | Just s => if %compatible "x" s tl then Just (%cons (("x",tl),s)) else Nothing ())
                    else Nothing ()
             | Const n => Nothing ()
             | Var s => Nothing ()) of
        Nothing () => t1
       | Just s => %subst s In Var "x"))>
```

Figure 4: Code generated from rule $r_1 = x + 0 \rightarrow x$ by function `match`.

but because we are forced to annotate `msigma` to have type: $<$`((string * Term)List)Maybe,'a`$>$ and not `((string *` $<$`Term,'a`$>$`)List)Maybe`, these two functions cannot "access" the list at generation time.

That our first attempt at staging our example was only partly successful, should not be too surprising. Users of partial evaluation systems restructure their programs to help BTA succeed all the time. Just because binding-time annotations are placed manually shouldn't exempt us from this requirement. Because we can touch, see, and experiment with both the explicit annotations and the code returned, it helps us understand and reason about what is going on. Using the type system to filter out obviously incorrectly phased programs was also extremely useful. The strength of MetaML is the mental model it provides to reason about what is going on.

A.2 Continuation-Passing Style

Our solution needs to propagate the context (performing a substitution over the right-hand side of the rule) into the leaves of the nested case expressions. This suggests rewriting the source program in continuation-passing style (CPS). This has been found to be quite useful in partial evaluation systems [1].

Figure 5 shows both a source version of `match` using an explicit continuation and an annotated two stage version of the same function. The function takes a pattern `pat`, a continuation `k`, a substitution `msigma`, and a term `term`. This function actually has a rather polymorphic type since nothing constrains the value returned by the continuation `k`. In the figure we have constrained the continuation to have type substitution to term. The continuation should apply the substitution to the right-hand side of the rule, or return the term unchanged if the substitution is a failure. Thus the wrapper function could be written:

```
fun wrapper (lhs,rhs) term =
```

```
(* Source *)
type Cont = Sub -> Term


(*  match : Term -> Cont -> Sub -> Term -> Term *)
fun match pat k msigma term =
case (msigma) of
   Nothing () => k (Nothing())
 | Just (sigma) =>
(case (out pat) of
    Var u =>
      if compatible u sigma term
         then k (Just (cons((u,term),sigma)))
        else k (Nothing ())
  | Op (t11,s1,t12) =>
      (case (out term) of
         Op (t21,s2,t22) =>
        (if streq(s2,s1)
            then (match t11
                    (fn s => match t12 k s t22)
                       msigma
                         t21)
            else k (Nothing ()))
        | _ => k(Nothing ()))
  | Const n =>
      (case (out term) of
         Const u => if u=n
                     then k msigma
                     else k (Nothing ())
       | _ => k(Nothing ())));
```

```
(* Annotated *)
type Sub' = ((string * <Term>) List) Maybe;
type Cont' = Sub' -> <Term>

(* match : Term -> Cont' -> Sub' -> <Term> -> <Term> *)
fun match pat k msigma term =
case (msigma) of
   Nothing () => k (Nothing())
 | Just (sigma) =>
(case (out pat) of
    Var u =>
     <if ~(compatible' u sigma term)
         then ~(k (Just (cons((u,term),sigma))))
         else ~(k (Nothing ()))>
  | Op (t11,s1,t12) =>
      <case (out ~term) of
         Op (t21,s2,t22) =>
       (if streq(~(lift s1),s2)
           then ~(match t11
                   (fn s => match t12 k s <t22>)
                      msigma
                        <t21>)
           else ~(k (Nothing ())))
       | _ => ~(k(Nothing ()))>
  | Const n =>
      <case (out ~term) of
         Const u => if u= ~(lift n)
                     then ~(k msigma)
                     else ~(k (Nothing ()))
     | _ => ~(k(Nothing ()))>);
```

Figure 5: Normal and Annotated versions of the CPS style `match`.

```
  match lhs
      (fn Nothing () => term | Just s => subst rhs s)
          (Just [])
              term;
```

The annotated version is again remarkably similar to the original except for the annotations. The only difference is that we needed to write a staged version of the `compatible` function, since in the staged version a substitution (`Sub'`) maps a string to a piece of code with type term, rather than a term. We call this function `compatible'`. It returns a piece of code with type `bool`.

```
fun compatible' u sigma term =
case find u sigma of
  Nothing() => <true>
| Just w => <termeq ~w ~term>;
```

Finally, the code generator can be constructed by supplying a suitable substitution continuation. This function needs a annotated substitution function which returns a piece of code with type term rather than a term, and is given below:

```
fun subst' (In t) sig =
case (t) of
  Var v =>
      (case find v sig of
         Nothing _ => <In (Var ~(lift v))>
       | Just w => w)
| Op (t1,s,t2) =>
      <In (Op (~(subst' t1 sig),
               ~(lift s),
               ~(subst' t2 sig))) >
| Const i => <In (Const ~(lift i))>;

fun wrapper (lhs,rhs) term =
  match lhs
      (fn Nothing () => term | Just s => subst' rhs s)
          (Just [])
              term;
```

When this function is used to generate code for the rule $r_1 = \ x + 0 \rightarrow x$ the following is generated:

```
<fn t =>
    (case %out t of
       Op(tl,s,tr) =>
          if %streq ("+",s)
             then (case %out tr of
                      Const n => if %= (n,0) then tl else t
                    | Var s => t
                    | Op z => t)
             else t
     | Const n => t
     | Var z => t)>
```

In addition to being compact and free of reducible nested `case`-statements or calls to `subst` or `compatible`, this output is virtually identical to the idealized code we presented for `rewriteR1`. We have observed that the code generated for a variety of other rules is equally as compact.