

From Interpreter to Compiler using Staging and Monads

Tim Sheard and Zine-el-abidine Benaïssa
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000 Portland, Oregon 97291-1000 USA

April 14, 1998

Abstract

In writing this paper we had two goals. First, to promote METAML, a programming language for writing staged programs, and second, to demonstrate that staging a program can have significant benefits. We do this by example: the derivation of an executable compiler for a small language. We derive the compiler in a rigorous fashion from a semantic description of the language. This is done by staging a denotational semantics, expressed as a monadic interpreter. The compiler is a program generator, taking a program in the source language (a *while-program*) as input and producing an ML program as target. The ML program produced is in a restricted subset of ML over which the programmer has complete control. It is encapsulated in a special data-structure called *code*. The meta-programming capabilities of METAML allow this data-structure to be directly executed (run-time code generation), or to be analysed. We illustrate this analysis of generated code to build a source to source transformation which applies the monad laws to significantly improve the generated code.

1 Compilers as staged interpreters

Interpreters, when implemented in high-level declarative languages, are very close to the interpreted language's denotational semantics. Because of this, interpreters are usually used for the development of prototypes, but such prototypes lack both efficiency and any connection to the underlying system in which the compiled code must run. If expressed in a monadic style, an interpreter can be mapped closer to the underlying system, and the structuring properties of the monad even allow the interpreter to be reused as the system evolves [32, 14, 27]. Nevertheless, the effort used to build the interpreter is often considered wasteful since the programmer still needs to re-implement the compiler from scratch after building the interpreter.

Our solution to this problem is the following multi-step method. First, construct the denotational semantics as an interpreter in a functional language. Second, capture the effects of the language, and the environment in which the target language must run, in a monad. Then rewrite the interpreter in a monadic style. Third, stage the interpreter using meta-programming techniques. This staging is similar to the staging of interpreters using a partial evaluator, but is explicit rather than implicit, since the programmer places the annotations directly, rather than using an automatic binding time analysis to discover where they should be placed. This leaves programmers in complete control, and they can limit what appears in the residual program. Fourth, the resulting program is both a data-structure and a program, so it can be both directly executed and analysed. This analysis can include both source to source transformations, or translation into another form (i.e.

intermediate code or assembly language). Because the programmer has complete control over the structure of the residual program this can be a trivial task.

Staging of interpreters using partial evaluation has been done before [2, 4]. The contribution of this paper is to show that this can all be done in a single program. A system incorporating staging as a first class feature of a language is a powerful tool. While using such a tool to write a compiler the source language can be given semantics, it can be staged, translated, and optimized all in a single paradigm. It requires neither additional processes nor tools, and is under the complete control of the programmer; all the while maintaining a direct link between the semantics of interpreter and those of the compiler. Staging organizes the task of constructing a compiler into simple, incremental steps, where the semantic connection is maintained through each stage of the derivation. Each step is a relatively easy task compared to building a compiler from scratch. Constructing a compiler using a staged language has the following benefits:

- **Simplicity.** Each task is a simple one, and builds incrementally on the previous tasks.
- **Correctness.** The compiler remains connected to its semantics. Each artifact produced by a task, is provably correct with respect to the artifacts of the previous tasks. The final artifact is both a compiler for the language and a semantics equivalent to the original semantics.
- **Reuse.** Each artifact reuses the code of the previous artifact.
- **Control.** The programmer has complete control over the resulting output. He develops his program with staging in mind, and the completely controls the structure of the residual program.

2 Staging in METAML

METAML is almost a conservative extension of Standard ML. Its extensions include four staging annotations. To delay an expression until the next stage one places it between meta-brackets. Thus the expression `<23>` (pronounced “bracket 23”) has type `<int>` (pronounced “code of int”). We illustrate the important features of the staging annotations in the short METAML session below.

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

-| val quad = ( 3+4, <3+4>, lift (3+4), <z> );
val quad = ( 7, <3 %+ 4>, <7>, <%z> ) :
           ( int * <int> * <int> * <int> )

-| fun inc x = <1 + ~x>;
val inc = Fn : ['a].<int> -> <int>

-| val six = inc <5>;
val six = <1 %+ 5> : <int>

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

Users access METAML through a *read-type-eval-print* top-level. The declaration for `z` is read, typed to see that it has a consistent type (`int` here), evaluated (to 7), and then both its value and type are printed.

The declaration for `quad` contrasts normal evaluation with the three ways objects of type code can be constructed. Placing brackets around an expression (`<3+4>`) defers the computation of `3+4` to the next stage, returning a piece of code. Lifting an expression (`lift (3+4)`) evaluates that expression (to 7 here) and then lifts the value to a piece of code that when evaluated returns the same value. Brackets around a free variable (`<z>`) creates a new constant piece of code with the value of the variable. Such constants print with a `%` sign to indicate they are constants. We call this *lexical-capture* of free variables. Because in METAML operators (such as `+` and `*`) are also identifiers, free occurrences of operators often appear with `%` in front of them.

The declaration of the function `inc` illustrates that larger pieces of code can be constructed from smaller ones by using the escape annotation. Bracketed expressions can be viewed as *frozen*, i.e. evaluation does not apply under brackets. However, it is often convenient to allow some reduction steps inside a large frozen expression while it is being constructed, by “splicing” in a previously constructed piece of code. METAML allows one to *escape* from a frozen expression by prefixing a sub-expression within it with the tilde (`~`) character. Escape must only appear inside brackets.

In the declaration for `six`, the function `increment` is applied to the piece of code `<5>` constructing the new piece of code `<1 %+ 5>`.

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside.

3 Monads in METAML

We assume the reader has a working knowledge of monads[30, 33]. We use the *unit* and *bind* formulation of monads[32]. In METAML a monad is a data structure encapsulating a type constructor M and the *unit* and *bind* functions.

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

This definition uses SML’s postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument (`'M : * -> *`) of the type constructor `Monad` is itself a unary type constructor [8]. We say that `'M` has *kind*: `* -> *`. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions [21]. The type variables in brackets, e.g. `['a, 'b]`, are universally quantified. Because of the explicit type annotations in the `datatype` definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the METAML type inference engine.

In METAML, `Monad` is a first-class, although *pre-defined* or *built-in* type. In particular, there are two syntactic forms which are aware of the `Monad` datatype: `Do` and `Return`. `Do` and `Return` are METAML’s syntactic interface to the *unit* and *bind* of a monad. We have modeled them after the `do`-notation of Haskell[10, 24]. An important difference is that METAML’s `Do` and `Return` are both parameterized by an expression of type `'M Monad`. Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. `Do` and `Return` are syntactic sugar for the following:

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

In addition the syntactic sugar of the `Do` allows a sequence of $x_i \leftarrow e_i$ forms, and defines this as a nested sequence of `Do`'s. For example:

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

The monad laws, expressed in METAML's `Do` and `Return` notation are:

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

4 The three-step method for compiler development

In this section, we illustrate our method by building the front end of a compiler for a small imperative *while-language*. We proceed in three steps. First, we introduce the language and its denotational semantics by giving a monadic interpreter as a one stage METAML program. Second, we stage this interpreter by using a two stage METAML program in order to produce a compiler. Third, we illustrate the usefulness of the staging approach, by defining a function that takes the output code of the compiler as input and returns an optimized version. This function is simply a pattern-matching based implementation of the monadic identity and associativity laws. This makes a dramatic difference in the quality of the generated code, and is completely reusable because the laws hold for any monad, not just the monad used in the example.

This illustrates the usefulness of combining the monadic and staged approaches. Without the monadic structure of the interpreter, the usefulness of the monadic-laws would have to be re-captured in a domain specific manner for every compiler. Without the structure provided by the staging, the pattern-matching based rewrite system would be impossible to use, because the compile-time computations would intervene and make recognition of the patterns impossible. In the staged interpreter, the compile-time code has disappeared by the time we want to apply the pattern based monadic-law transformer.

4.1 The while-language

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

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

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

```

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

```

A simple while-program in concrete syntax, such as

```

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

```

is encoded abstractly in these datatypes as follows:

```

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

```

4.2 The structure of the solution

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

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

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

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

Splitting the environment into two components is a standard technique (often called a binding time improvement) used by the partial evaluation community[9]. We capture this precisely by the following purely functional implementation.

```

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

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

```

```

    in pos 1 index end;

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

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

```

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

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

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

```

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

```

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

```

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

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

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

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

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

```

4.3 Step 1: monadic interpreter

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

The only non-standard morphism evident in the `eval1` function is `read`, which describes how the value of a variable is obtained. The monadic interpreter for expressions

takes an index mapping names to locations and returns a computation producing an integer.

```
(* eval1: Exp -> index -> int M *)
fun eval1 exp index =
case exp of
  Constant n => Return msw0 n
| Variable x => let val loc = position x index
                in read loc end
| Minus(x,y) =>
  Do msw0 { a <- eval1 x index ;
            b <- eval1 y index;
            Return msw0 (a - b) }
| Greater(x,y) =>
  Do msw0 { a <- eval1 x index ;
            b <- eval1 y index;
            Return msw0 (if a >> b then 1 else 0) }
| Times(x,y) =>
  Do msw0 { a <- eval1 x index ;
            b <- eval1 y index;
            Return msw0 (a * b) };
```

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

```
(* interpret1 : Com -> index -> unit M *)
fun interpret1 stmt index =
case stmt of
  Assign(name,e) =>
    let val loc = position name index
        in Do msw0 { v <- eval1 e index ; write loc v } end
| Seq(s1,s2) =>
  Do msw0 { x <- interpret1 s1 index;
            y <- interpret1 s2 index;
            Return msw0 () }
| Cond(e,s1,s2) =>
  Do msw0 { x <- eval1 e index;
            if x=1
              then interpret1 s1 index
              else interpret1 s2 index }
| While(e,body) =>
  let fun loop () =
        Do msw0 { v <- eval1 e index ;
                  if v=0 then Return msw0 ()
                    else Do msw0 { interpret1 body index ;
                                    loop () } }
      in loop () end
| Declare(nm,e,stmt) =>
  Do msw0 { v <- eval1 e index ;
            push v ;
            interpret1 stmt (nm::index);
            pop }
| Print e =>
  Do msw0 { v <- eval1 e index;
            output v };
```

Although `interpret1` is fairly standard, we feel that two things are worth pointing out. First, the clause for the `Declare` constructor, which calls `push` and `pop`, implicitly

changes the size of the stack and explicitly changes the size of the index (`nm:index`), keeping the two in synch. It evaluates the initial value for a new variable, extends the index with the variables name, and the stack with its value, and then executes the body of the `Declare`. Afterwards it removes the binding from the stack (using `pop`), all the while implicitly threading the accumulated output. The mapping is in scope only for the body of the declaration.

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

```
| While(e,body) =>
  Do msw0 { v <- eval1 e index ;
            if v=0 then Return msw0 ()
              else Do msw0 { interpret1 body index ;
                            interpret1 (While(e,body)) index }
          }
}
```

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

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

4.4 Step 2: staged interpreter

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

```
(* eval2: Exp -> index -> <int M> *)
fun eval2 exp index =
case exp of
  Constant n => <Return msw0 ~(lift n)>
| Variable x =>
  let val loc = position x index
      in <read ~(lift loc)> end
| Minus(x,y) =>
  <Do msw0 { a <- ~(eval2 x index) ;
            b <- ~(eval2 y index);
            Return msw0 (a - b) }>
| Greater(x,y) =>
  <Do msw0 { a <- ~(eval2 x index) ;
            b <- ~(eval2 y index);
```

```

      Return msw0 (if a '>' b then 1 else 0) }>
| Times(x,y) =>
  <Do msw0 { a <- ~(eval2 x index) ;
            b <- ~(eval2 y index);
            Return msw0 (a * b) }>;

```

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

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

```

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

< Do msw0
  { a <- ~(eval2 (Constant 3) []);
    b <- ~(eval2 (Constant 1) []);
    Return msw0 (a-b)} > =

< Do msw0
  { a <- ~<Return msw0 3>;
    b <- ~<Return msw0 1>;
    Return msw0 (a - b)} > =

< Do msw0
  { a <- Return msw0 3;
    b <- Return msw0 1;
    Return msw0 (a - b)} > =

< Do %msw0
  { a <- Return %msw0 3;
    b <- Return %msw0 1;
    Return %msw0 (a %- b)} >

```

Each recursive call produces a bracketed piece of code which is spliced into the larger piece being constructed. Recall that escapes may only appear at level-1 and higher. Splicing is axiomatized by the reduction rule: $\sim\langle x \rangle \longrightarrow x$, which applies only at level-1. The final step, where `msw0` and `-` become `%msw0` and `%-`, occurs because both are free variables and are lexically captured.

Now we can state the equivalence relationship between the monadic `eval1` and the staged `eval2`. We use the axiomatic semantics of METAML [28], in particular the axioms for the annotations, such as the splice axiom above.

Proposition 1. For all expressions `exp`, and list of names `index`:

```
eval1 exp index = run (eval2 exp index)
```

Proof. We might argue that there is a trivial proof to this proposition. Since `eval1` is simply a copy of `eval2` with all the staging annotations erased, and that both functions type-check, by the semantics of METAML they must be equal. We include a more traditional proof in the appendix using the axiomatic semantics of METAML [28] (see appendix A).

Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

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

4.4.1 An example.

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

```
<Do %mswo
  { a <- Return %mswo 10
  ; %push a
  ; Do %mswo
    { e <- Do %mswo
      { d <- Do %mswo
        { b <- %read 1
        ; c <- Return %mswo 1
        ; Return %mswo b %- c
        }
      ; %write 1 d
      }
    }
  ; g <- Do %mswo
```

```

        { f <- %read 1
          ; %output f
        }
      ; Return %mswo ()
    }
  ; %pop
}>

```

Note that the staged program is essentially a compiler, translating the syntactic representation of the while-program into the above monadic object-program that will compute its meaning. This program sequentializes the decrement x and the print of x . This object-program is fully executable. Simply by using the `run` operator of METAML, it can be executed for prototyping purposes.

Equally important, the object-program itself is just a piece of data, which can be analyzed and further translated in another layer of the translation pipeline. The reader might notice that this object-program could be further simplified by applying the monad laws. There are many opportunities for doing so. After these laws are applied we obtain the much more satisfying:

```

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

```

In addition to the monad laws which hold for all monads, we can also use laws which hold for particular non-standard morphisms. For instance, in the example above, we could avoid the second read of location 1 using the following rule:

$$\text{Do } \{ e1; c \leftarrow \%write\ 1\ b ; d \leftarrow \%read\ 1; e2 \} = \text{Do } \{ e; c \leftarrow \%write\ 1\ b; e2[b/d] \}$$

Every target language will have many such laws, and because our target language is both executable-code, and data-structure we can perform these optimizations. How this is accomplished is the subject of Section 4.5.

As for the `eval` function, we state the semantic equivalence between the monadic and the staged interpreters.

Proposition 2. For all commands `com` and list of names `index`:

$$\text{interpret1 com index} = \text{run (interpret2 com index)}$$

Proof. See appendix A.

4.5 Step 3: optimizing target code: the monadic laws

Perhaps the most important contribution we make in this paper, is that a staged program produces a piece of code that is both an executable-program and a data-structure.

If one wants to execute this code, one uses the `run` annotation. If one wants to optimize this code, this is possible as well. In this section we illustrate this by example; providing an implementation of the monad law transformations demonstrated in section 4.4.1

In this section, we briefly explain our method for analysing (or computing over, or doing intensional analysis of) METAML code. We believe, that operations such as pattern-matching and substitution on code should be provided once and for all by the system, and not by the user.

Optimizations are generally thought of as rewriting rules or transformations. Both the rules and the strategy (e.g. top-down or bottom-up) needed to apply them need to be described.

To illustrate this point, we write a simple transformation which implements the monadic laws as directed rewrites. As a reminder, the monadic laws expressed in terms of METAML's `Do` and `Return` notation are repeated.

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

To implement these rules, we need a mechanism for pattern matching over code. Like all METAML code, the result of the monadic interpreter is just a data structure so this is possible.

Let us consider a simple example. Suppose we want to match all pieces of code that are of the form `<Δ + 3>`. We have used the `Δ` to indicate a meta-variable that will match any piece of code. We cannot put a variable (e.g. `x`) here because `<x+3>` is just a piece of code and not a pattern. The solution to indicating a meta-variable in a pattern is to use an escaped variable at level-1 in the pattern. Thus the pattern `<~x + 3>` matches all pieces of code that have this “shape”.

Unfortunately, this scheme is not always sufficient when matching against code with binding constructs such as `<fn x => x + 1>`. We would like to construct a pattern that matches against a function (or other binding construct) and to be able to use the meta-variables bound inside the pattern to construct a transformation. To see why this is problematic consider the following two examples:

1. We want a transformation that increments the body of an integer valued function, such that when applied to `<fn x => x>` we obtain `<fn x => x + 1>`, and when applied to `<fn y => length y>` we obtain `<fn y => (length y) + 1>`. As a first approximation we try: `<fn x => Δ> => <fn x => Δ + 1>`. This looks promising, but what would happen if we wrote: `<fn x => Δ> => <fn y => Δ + 1>` instead? Now, free occurrences of `x` in `Δ` no longer have a binding site, because they have been spliced into a context where `y` is the bound variable instead of `x`.
2. We want a transformation that doubles the argument of an `int -> int` function, such that when applied to `<fn x => x>` we obtain `<fn x => x + x>` and when applied to `<fn x => y + x>` we obtain `<fn x => y + (x + x)>`. The problem here is that in the pattern, `<fn x => Δ>`, there is no way to express that `Δ` may have free occurrences of `x` inside, and that our transformation needs to substitute for those free occurrences.

The solution is to use a higher-order pattern. Suppose we could parameterize `Δ` on `x`. This makes `(Δx)` not a meta-variable with type code, but a meta-variable with type code to code. Inside a pattern on the left hand side of a match (`pat => exp`) a higher order meta-variable is bound to a function when it is successfully matched against a piece of code. On the right hand side of the match, when this meta-variable is used (by applying it to a piece of code) it substitutes all occurrences of `x` with the argument it was applied to. For example consider the table below showing the binding of the higher order meta variable `Δx` when the pattern `<fn x => Δx + 3>` is matched against different pieces of code.

code matched against	function bound to
<code><fn x => x + 3></code>	<code>fn x => < ~x ></code>
<code><fn x => (x - 9) + 3></code>	<code>fn x => < ~x - 9 ></code>
<code><fn x => (sin x + x^2) + 3></code>	<code>fn x => < sin ~x + ~x^2 ></code>
<code><fn x => x + 1></code>	match failure

To express this in METAML we use the convention that the function in an escaped application (where all the arguments of the application are explicitly bracketed code) represents a higher order meta-variable. Thus, whenever an escaped application appears inside a pattern, the function part of the application is a higher-order meta-variable and its arguments are its formal parameters. For example: `~(g <x>)`. The two problematic examples above are now easily expressed as:

```
<fn x => ~(g <x>)> => <fn y => ~(g <y>) + 1>
<fn x => ~(h <x>)> => <fn z => ~(g <z + z>)>
```

Because higher order meta-variables may appear only in the function position of escaped applications, and the arguments of these escaped applications may only be bracketed bound variables (like `<x>`), pattern-matching and unification are decidable [16, 25].

We now possess the tools to present the monad-law and code-optimizing METAML-function `opt`:

```
fun opt < Do ~st {x <- ~v ; return ~st' x } > = opt v
| opt w as < Do ~st {x <- Return ~st' ~e ; ~(z <x>) > =
    if is_constant e then opt (z e) else w
| opt < Do ~st {x <- Do ~st' {y <- ~e ; ~(f <y>)} ; ~(g <x>)} > =
    opt <Do ~st {y' <- ~e ; x' <- ~(f <y'>) ; ~(g <x'>)}>
| opt x = map_code opt x (* traversal through the code *)
```

Our `opt` function implements a limited form of the *left-id* monad law. We do not wish to duplicate by substitution a non-constant. By composing this optimization with `interpret2` we obtain a better compiler. Applying this compiler to:

```
Declare x = 150 in
  Declare y = 200 in while x > 0 Do { x := x - 1; y := y - 1 }
```

we obtain following program:

```
<Do %state
{ a <- %push 150;
  b <- %push 200;
  c <- let fun loop () =
      Do %state
      { e <- %read 1;
        f <- return %state (if (e %> 0) then 1 else 0);
        if (f %= 0)
          then return %state ()
          else Do %state
              { g <- %read 1;
                h <- return %state (g - 1);
                i <- %write 1 h;
              }
      }
  }>
```

```

                                j <- %read 0;
                                k <- return %state (j - 1);
                                l <- %write 0 k;
                                loop ()
                            }
                    }
            in loop () end
    m <- %pop;
    %pop
} >
>

```

The optimizer has fully sequentialized the code using the bind-associativity law, and removed all superfluous `Return`'s using the unit-identity laws. Further optimizations, such as arithmetic simplification, or transformations to another form, such as assembly code, could be implemented in the same fashion.

5 Related work

Our work was inspired by work in many different areas. Derivation of compilers from specifications and the use of action-semantics [19, 23, 11, 22]; the use of monads to structure programs in general [18, 31, 26] and language implementations in particular [32, 27, 14]; staged programming [5, 6] and its use in structuring compilers [29, 20, 4]; partial evaluation [34, 17, 1, 3, 2, 9]; higher order abstract syntax and pattern matching [16, 7]

For space considerations we limit detailed discussion to the following areas.

5.1 Monads and compilation

Perhaps, the most related work is the work of Sheng Liang and his thesis advisor Paul Hudak [12, 13]. They investigate the derivation of a compiler from a modular monadic interpreter. Our work is a continuation of their effort of using monads as a standard compilation mechanism. However, some differences remain:

- The use of staging, lead us at an early step in the development, to split the environment into a static index of names and a dynamic stack of values. This allows us to avoid the use of an environment monad. We use instead an state transformer monad in which the state is managed like a stack. Liang uses a complicated monad which is a combination of an environment monad and a state transformer. After code generation they show that the residual code due to the environment (the lookups of the location of variables) can be eliminated using axioms of the non-standard morphisms of the environment monad. Our use of staging allowed us to do the lookups in the first stage and to never residualize the lookups at all.
- On the other hand, Liang's use of modular language components is an advantage we have not even attempted to employ. For simplicity, we have used the same monad for both expressions and commands while Liang uses a modular approach where each feature is defined independently from the others. Finally all the features are combined by a monad transformer. To do this it is necessary to lift all non-standard morphisms through the transformer. This is hard and not completely understood. We may try to duplicate Liang's approach in future work.

5.2 Staging and compilation

In his thesis *Calculating Compilers* [15] Erik Meijer advocates staging a compiler by using self discipline. Construct a compiler by building it as the composition of compile-time and run-time components. A critical step in this process is finding a representation of every source language construct as a combination of (lower level) target level constructs. By representing both source and target languages as algebraic datatypes, say `source` and `target`, induced by the functors `S` and `T`, this can be reduced to finding a polymorphic function `Trans`, which for all α , has type $(T\alpha \rightarrow \alpha) \rightarrow (S\alpha \rightarrow \alpha)$, a so-called algebra transformer.

Let the semantic domain of the target algebra be some type `value`. If the semantic meaning function for the target language `M:target -> value` can be expressed as a catamorphism `M = cata phi` where `phi:T value -> value`, we can lift `phi` into an interpreter for the source language by applying the algebra transformer `Trans`. Thus `Trans phi:S value -> value` and `Interp = cata (Trans phi):source -> value`. A similar construction can be used to construct the compiler `Compiler:source -> target`. Let function `In:T target -> target` be the injection between the functor `T` and its induced algebraic datatype `target`, then `cata (Trans In):source -> target` constructs the compiler.

The limiting factor in this approach is finding an algebraic datatype `target` to encode the target language. For a monadic target language, it is not known how to do this, since the constructors for “unit” and “bind” would be too polymorphic to encode in an algebraic datatype, and many of the non-standard morphisms would not be polymorphic enough.

By staging the process in METAML, we do away with the need for an algebraic datatype to encode the target language, by using the special type of code instead. The constructors of the target algebra are simply the second stage representations of the real functions.

5.3 Difference between staging and partial evaluation

Staged programming (S.P.) is closely related to partial evaluation (P.E.). We list what we believe are the salient differences.

- S.P. uses explicit annotations while P.E. uses implicit annotations placed by an automatic binding time analysis.
- S.P. gives the programmer complete control over what residual program is produced, while the residual program produced by P.E. often contains surprises. The surprises are caused by the differences between what the programmer knows and what the binding time analysis can discover. The solution to this mismatch is for the programmer to restructure his program using “binding-time improvements” which more closely align his knowledge and the capabilities of the binding time analysis. Of course S.P. is not completely immune to these difficulties, but the staged programmer must be fully aware of the staging issues before he writes his program. The staged type-system is a great advantage here. Nevertheless, there are many simple programs where automatic binding time analysis is sufficient, and hand staging is simply an annoyance. In our system we have combined the advantages of both, allowing a simple type-directed binding time analysis to co-exist with the manual staging annotations. An analysis of this co-existence is beyond the scope of this paper.
- S.P. is a programming language feature. It exists at the same level as the program. Here the algorithm and the staging are developed hand in hand. There are no

additional tools or processes, and users learn how to weave the staging thought processes into their problem solving techniques.

- S.P. provides a complete, unified, typed environment, supporting both type reconstruction and polymorphism for the staged constructs.

6 The Implementation

Everything you have seen in this paper, except the higher order pattern matching over code, has been implemented in the METAML implementation. The examples are actual runs of the system.

The higher order pattern matching is currently under development. We found the normalizing effect of the monad laws¹ so compelling that we implemented them in an ad-hoc fashion inside the METAML system.

7 Conclusion

We have shown that staging programs offers an exciting new programming paradigm, and reinforced the notion that staging a monadic interpreter into compile-time and run-time components provides a direct link between an interpreter and a compiler.

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

References

- [1] Anders Bondorf and Olivier Danvy. Automatic autoprojectin of recursive equations with global variables and data types. *Science of Computer Programming*, 16:151–195, 1991.
- [2] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 308–320, New York, June 1993. ACM Press. Copenhagen.
- [3] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 145–154. New York: ACM, 1993.
- [4] O Danvy, J Koslowski, and K Malmkjaer. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 91.
- [5] 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.
- [6] 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.

¹as well as the laws for β -value, let-normalization, and η -value reduction

- [7] Carsten Schürmann, Frank Pfenning, Joëlle Despeyroux. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, number 1210 in LNCS, pages 147–163. Springer-Verlag, April 1997.
- [8] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.
- [9] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Series editor C. A. R. Hoare. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
- [10] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.
- [11] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.
- [12] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale university, 1998.
- [13] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP'96: 6th European Symposium on Programming*, number 1058 in LNCS, pages 333–343, Linköping, Sweden, January 1996.
- [14] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343, January 1995.
- [15] Erik Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
- [16] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen, Germany, December 1989*, volume 475 of LNCS, pages 253–281. Springer-Verlag, 1991.
- [17] Torben Mogenson. Self-applicable partial evaluation for the pure lambda calculus. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation*, pages 116–121, June 1992. Yale University Department of Computer Science Technical Report YALEU/DCS/RR-909.
- [18] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, July 1991.
- [19] Peter D. Mosses. SIS-semantics implementation system, reference manual and users guide. Technical Report DAIMI report MD-30, University of Aarhus, Aarhus, Denmark, 1979.
- [20] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, Mass., 1992.

- [21] Martin Odersky and Konstantin Läuffer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 54–67, January 1996.
- [22] Jens Palsberg. A proveably correct compiler generator. In B. Krieg-Bruckner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France*, pages 418–434, New York, February 1992. Springer Verlag. Lecture Notes in Computer Science 582.
- [23] L. Paulson. *Methods and Tools for Compiler Construction, B. Lorho (editor)*. Cambridge University Press, 1984.
- [24] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.
- [25] Z. Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT'93*, volume 668 of *LNCS*, pages 391–405. Springer-Verlag, 1993.
- [26] Michael Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.
- [27] Guy Steele. Building interpreters by composing monads. In *21st Annual ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
- [28] Walid Taha, Zine-El-Abidine Benaïssa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In *International colloquium on automata, languages, and programming*, LNCS. Springer-Verlag, July 1998. To appear.
- [29] William L. Scherlis Urlik Jorring. Compilers and staging transformations. In *13th ACM Symposium on Principles of Programming Languages*, pages 86–96. ACM, ACM Press, January 1986.
- [30] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.
- [31] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.).
- [32] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.
- [33] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.
- [34] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference*, pages 165–191, Cambridge, Ma, USA, August 1991. Springer Verlag. Lecture Notes in Computer Science 523.

A Proofs

We repeat here the axiomatic semantics of METAML [28]. For the sake of simplicity, we omit the level-annotations.

$$\begin{array}{lll}
 \text{run } \langle v1 \rangle & = v1 \downarrow & (\text{run}) \\
 \sim \langle e \rangle & = e & (\text{escape}) \\
 (\lambda x. e) v & = e[x := v] & (\text{beta})
 \end{array}$$

The (escape) axiom applies only at level one (inside exactly one bracket) and (run) and (beta) apply only at level 0 (inside no brackets).

Lemma 1. *For any well-typed expression: $\langle \sim e \rangle$, we have $\langle \sim e \rangle = e$*

Proof. Since $\langle \sim e \rangle$ is well-typed, e must evaluate (if it terminates) to $\langle v \rangle$. Then $e = \langle v \rangle$. We have

$$\begin{array}{ll}
 \langle \sim e \rangle = \langle \sim \langle v \rangle \rangle & \text{replace equals by equals} \\
 = \langle v \rangle & \text{By escape axiom} \\
 = e &
 \end{array}$$

□

Lemma 2. *For any well-type expression: $\text{run } \langle f e \rangle$, we have*

$$\text{run } \langle f e \rangle = (\text{run } \langle f \rangle) (\text{run } \langle e \rangle).$$

Proof. Since the term $f e$ is at level 1, the only possible reduction is by the escape axiom. Assume $\langle f \rangle$ and $\langle e \rangle$ evaluate to the values $\langle f1 \rangle$ and $\langle e1 \rangle$ respectively. Then $\langle f e \rangle$ must evaluate to $\langle f1 e1 \rangle$ (since at level 1 we cannot do a beta-step). Hence, we have $\langle e \rangle = \langle e1 \rangle$, $\langle f \rangle = \langle f1 \rangle$, $\langle f e \rangle = \langle f1 e1 \rangle$

$$\begin{array}{ll}
 \text{run } \langle f e \rangle = \text{run } \langle f1 e1 \rangle & \text{by replacing equals by equals} \\
 = (f1 e1) \downarrow & \text{by run axiom} \\
 = (f1 \downarrow) (e1 \downarrow) & \text{by definition of } \downarrow \\
 = (\text{run } \langle f1 \rangle) (\text{run } \langle e1 \rangle) & \text{by run axiom} \\
 = (\text{run } \langle f \rangle) (\text{run } \langle e \rangle) & \text{by replacing equals by equals}
 \end{array}$$

□

Lemma 3. *For any well-type expression: $\text{run } \langle \lambda x. e \rangle$, we have*

$$\text{run } \langle \lambda x. e \rangle = \lambda x. (\text{run } \langle e \rangle).$$

Proof. The proof is similar to the two lemmas above. □

A consequence of the previous two lemmas is that `run` distributes through its sub-expressions. In particular, `run` distributes through `Do` and `let`.

$$\text{run } \langle \text{Do } \{ x1 \leftarrow e1 ; x2 \leftarrow e2 ; \dots ; en \} \rangle = \text{Do } \{ x1 \leftarrow \text{run } \langle e1 \rangle ; x2 \leftarrow \text{run } \langle e2 \rangle ; \dots ; \text{run } \langle en \rangle \} \quad (\text{run-Do})$$

$$\text{run } \langle \text{let val } x = e \text{ in } e2 \rangle = (\text{let val } x = \text{run } \langle e1 \rangle \text{ in } \text{run } \langle e2 \rangle) \quad (\text{run-Let})$$

Proposition 1. *For all expressions exp , and list of names index :*

$$\text{eval1 exp index} = \text{run } (\text{eval2 exp index})$$

Proof. Induction on the structure of `exp`.

case `exp` of `Minus(e1,e2)`

```
run (eval2 (Minus(e1,e2)) index) = By beta axiom

run <Do msw { a <- ~ (eval2 e1 index) ;
              b <- ~ (eval2 e2 index) ;
              Return msw (a -b) } > = by (run-Do)

Do msw { a <- run <~(eval2 e1 index)> ;
         b <- run <~(eval2 e2 index)> ;
         run <Return msw (a-b)> } = by lemma1 (twice) and run axiom

Do msw { a <- run (eval2 e1 index) ;
         b <- run (eval2 e2 index) ;
         Return msw (a-b) } = by induction hypothesis (twice)

Do msw { a <- eval1 e1 index ;
         b <- eval1 e2 index ;
         Return msw (a-b) } = by beta
```

`eval1 (Minus(e1,e2)) index`

The other cases are similar. □

Proposition 2. *For all commands `com` and list of names `index`:*

```
interpret1 com index = run (interpret2 com index)
```

Proof. By induction on the structure of `com`.

case `com` of `While(e,body)`.

```
run (interpret2 (While(e,body)) index) = By beta
run <let fun loop () =
      Do msw { v <- ~(eval2 e index);
              if v=0
                then Return msw ()
                else Do msw { q <- ~(interpret2 body index); loop ()}
            }
    in loop () end > = by run-Do and run-Let

=

let fun loop () =
      Do msw { v <- run <~(eval2 e index)>;
              if v=0
                then run <Return msw ()>
                else Do msw { q <- run <~(interpret2 body index)>;
                              run < loop () >}
            }
    in run < loop () > end

= By Lemma 1 and run axiom
```

```

let fun loop () =
  Do msw0 { v <- run (eval2 e index)>;
    if v=0
    then Return msw0 ()
    else Do msw0 { q <- run (interpret2 body index);
      run < loop () >}
  }
in run < loop ()> end = By induction hypothesis and Proposition 1

```

```

let fun loop () =
  Do msw0 { v <- (eval1 e index)>;
    if v=0
    then Return msw0 ()
    else Do msw0 { q <- interpret1 body index); run <loop ()> }
  }
in run <loop ()> end = By run axiom

```

```

let fun loop () =
  Do msw0 { v <- (eval1 e index)>;
    if v=0
    then Return msw0 ()
    else Do msw0 { q <- interpret1 body index); loop () }
  }
in loop () end

```

The last step is only possible because, at this step in the derivation, there are no annotations (in particular no escapes) in the body of the function loop, thus the body of loop at level 1 is a value, and hence in normal form.

The other cases are easier. □