DSL Implementation Using Staging and Monads

Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000 Portland, Oregon 97291-1000 USA
sheard@cse.ogi.edu, http://cse.ogi.edu/~ sheard

Abstract

The impact of Domain Specific Languages (DSLs) on software design is considerable. They allow programs to be more concise than equivalent programs written in a high-level programming languages. They relieve programmers from making decisions about data-structure and algorithm design, and thus allows solutions to be constructed quickly. Because DSL's are at a higher level of abstraction they are easier to maintain and reason about than equivalent programs written in a high-level language, and perhaps most importantly *they can be written by domain experts rather than programmers.*

The problem is that DSL implementation is costly and prone to errors, and that high level approaches to DSL implementation often produce inefficient systems. By using two new programming language mechanisms, program staging and monadic abstraction, we can lower the cost of DSL implementations by allowing reuse at many levels. These mechanisms provide the expressive power that allows the construction of many compiler components as reusable libraries, provide a direct link between the semantics and the low-level implementation, and provide the structure necessary to reason about the implementation.

1 Introduction

We outline an improved *method* for the design and implementation of Domain-Specific Languages (DSLs). The method builds upon our experience with staged programming using the staged programming language MetaML [27, 26]. The method also incorporates ideas from other researchers in the areas of modular language design [28, 24, 12], correct compiler generation [15, 19, 18, 16, 10], and partial evaluation [8, 13]. While relying on recent advances in functional programming (such as higher-order type constructors, and local polymorphism), it is applicable to *all kinds of languages*, not just applicative ones. The method unifies many of these ideas into a coherent process.

A problem with the DSL approach to software construction is its cost. Realizing a DSL requires an implementation. Such implementations are large and expensive to produce. So, unless many solutions are required, it may not pay to build a compiler or other implementation mechanism. DSL implementation is also conceptually hard. Most software engineers are not comfortable taking on the task of language design and implementation. Even if they are, language implementation is a difficult, complex process that does not easily scale. An implementation for a simple language often does not scale as the language evolves to meet newer demands. Lowering the cost of DSL implementations, and making good ones more manageable, will make the DSL approach applicable to a broader domain of problems.

Our approach to solving these problems is to apply new methods of abstraction such as monads [28, 31] and staging [27, 26] to the implementation of DSLs. This makes the effort required to build a compiler for a DSL reusable and spreads the cost over several DSLs. To make language implementation manageable for the masses, there must exist good rules of thumb for language implementation. One way to accomplish this is by elaborating *a step by step method* that splits the labor into well-defined steps, each with a relatively small amount of work. In our method, each step deals with an orthogonal design decision. By using good abstraction principles, our method partitions each design decision into a separate code module. In addition, our method makes explicit the propositions that must be proved to show the correctness of the compiler with respect to its semantics.

Our method comprises the following steps. 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 analyzed. 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 earlier steps, the structure of the residual program is highly constrained, and this final translation can be a trivial task.

Staging of interpreters using partial evaluation has been done before [1, 5]. 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.

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"). The annotation, ~e splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and `run` e evaluates e to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that ~e is only legal within lexically enclosing Brackets. We illustrate the important features of the staging annotations in the short MetaML sessions below.

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

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

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

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 in constructed code often appear with `%` in front of them.

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

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, is it 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.

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

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

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

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside. To give a brief feel for how MetaML is used to construct larger pieces of code at run-time consider:

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

-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))>
         : <int  -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn  : int  -> <int  -> int>
```

The function `mult`, given an integer piece of code `x` and an integer `n`, produces a piece of code that is an `n`-way product of `x`. This can be used to construct the code of a function that performs the `cube` operation, or generalized to a generator for producing an exponentiation function from a given exponent `n`. Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [4, 6, 11, 23, 27]. In this paper we use staging to construct compilers from interpreters.

3 Monads in Langauge Design

We make significant use of the notion of monads. A good way to think of a monad is as an abstract datatype that captures the side effects and actions inherent in the language being translated in the methods of the abstract datatype. An important feature of a monad is that also describes, in a purely functional way, how these effects and actions interact. Like any good abstract datatype, we are free to implement the actions in any way we want as long as our implementation behaves like its purely functional description.

The ultimate efficiency of the compiler depends on making good use of the low-level primitives of the target language. Monads are the glue that we use to tie high-level (purely functional) descriptions of languages to the low-level implementation features of the target environment.

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

An action might perform I/O, update a mutable variable, or raise an exception. One can implement a monad in a purely functional setting by emulating the actions. This is done by explicitly threading "stores", "I/O streams", or "exception continuations" in and out of all computations. We call such an emulation the *reference implementation*. Using a functional implementation allows equational reasoning about the reference implementation, however it is usually quite inefficient.

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

$$
\begin{array}{lrl}
\text{(left id)} & bind\ (unit\ x)\ (\lambda y.e) = & e[x/y] \\
\text{(right id)} & bind\ e\ (\lambda y.unit\ y) = & e \\
\text{(bind assoc)} & \multicolumn{2}{l}{bind\ (bind\ e\ (\lambda x.f))\ (\lambda y.g) =} \\
 & \multicolumn{2}{r}{bind\ e\ (\lambda z.bind\ (f[z/x])(\lambda w.g[w/y]))}
\end{array}
$$

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

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

For more background on the use of monads see [29, 31, 30].

4 Monads in MetaML

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

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

This definition uses SML's postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument `('M : * -> * )` of the type constructor `Monad` is itself a unary type constructor [7]. We say that `'M` has *kind*: `* -> *`. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions [17]. 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[9, 20]. An important difference is that MetaML's `Do` and `Return` are both parameterized by an expression of type `'M Monad`. `Do` and `Return` are syntactic sugar for the following:

```
(* Syntactic Sugar              Derived Form *)

Do (Mon(unit,bind)) { x <- e; f }    =
                              bind e (fn x => f)

Return (Mon(unit,bind)) e          =    unit e
```

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

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

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

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

5 Illustrating our compiler development method

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 showing how using MetaML's intensional analysis tools can be used to optimize or further translate the output of a staged program.

5.1 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)
| Seq     of (Com * Com)
| Cond    of (Exp * Com * Com)
| While   of (Exp * Com)
| Declare of (string * Exp * Com)
| Print   of Exp;

(* ****** Example Concrete Syntax ******* *)
(* Assign    x := 1                       *)
(* Seq       { x :=1; y:= 2 }             *)
(* Cond      if x then x := 1 else y := 1 *)
(* While     while x>0 do x := x - 1      *)
(* Declare   declare x = 1 in x := x - 1  *)
(* Print     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"))));
```

5.2 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[8]. 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 (stack -> ('a * stack * 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);

(* mswo is the Monad of state with output *)
val mswo : M Monad = Mon(unit,bind);
```

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)^" "));
```

5.3 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 mswo n
| Variable x => let val loc = position x index
                  in read loc end
| Minus(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (a - b) }
| Greater(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (if a '>' b
                             then 1 else 0) }
| Times(x,y) =>
   Do mswo { a <- eval1 x index ;
```

```
          b <- eval1 y index;
          Return mswo (a * b) };
```

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

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

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

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

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

5.4 Step 2: staged interpreter

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

```
(* eval2: Exp -> index -> <int M> *)
```

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

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

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

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

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

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

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

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

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

Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

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

5.4.1 An example.

The function `interpret2` generates a piece of code from a `Com` datatype. 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. Note that in the object-program all of the compile-time operations have disappeared. This object-program is fully executable. Simply by using the `run` operator of MetaML, it can be executed for prototyping purposes.

6 Step 3: Back-end translation and intermediate code optimization

MetaML is a meta-programming system. It has an object language and a meta-language. Meta-programs are programs that manipulate object programs. In MetaML both the object language and the meta-language are ML. In MetaML an object-program is both a data structure that can be manipulated, and a program that can be run.

This duality plays an important role in target code generation. The result of applying the staged interpreter from the previous step (a meta-program) to a DSL program to be compiled is a highly constrained residual program (an object program). This program is both a data-structure and a program, so it can be both directly executed (rapid prototype) and analyzed.

We use the object-code analysis capabilities of MetaML to transform the object program into the final target language. This analysis can include both source to source transformations, or translation into another form (i.e. intermediate code, assembly language, or target language).

Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. For example, we may structure the staged interpreter such that the residual program is first order, or just a sequence of primitive actions encoded as non-standard morphisms in the monad. This is where we connect the abstract monadic actions to their efficient implementations.

The object program produced above is an ML code fragment. It can be executed or analyzed. The code produced by `interpret2` is a restricted subset of ML. Disregarding the higher-order functions implicit in the monad, it is first order, and contains only `Do` expressions, `Return` expressions, `if` expressions, calls to the non-standard morphisms `read`, `write`, `push` , `pop`, and `output`, primitive arithmetic operators `-` and `'>'`, and local looping functions (like `loop` above). The code is so regular that it can be captured by a simple grammar. The next step is to analyze this code to make the final translation to the target language, or to apply some ML-source to ML-source level optimizations. The reader might notice that the object-program above could be considerably, 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:

```
Do { e1
   ; c <- %write 1 b
   ; d <- %read 1; e2
   }
=
Do { e
   ; c <- %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. The final step is to translate the ML code fragment into the target language. This step uses the same intensional analysis of code capabilities of the optimization steps, and is the subject of the next section.

6.1 Intensional analysis of code fragments

In this section, we outline how we do intensional analysis of residual code. We provide a high-level pattern matching based interface. Code patterns can be constructed by placing brackets around code. For example a pattern that matches the literal `5` can be constructed by:

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

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

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

The function `is5` matches its argument to the constant pattern `<5>` if it succeeds it returns `true` else `false`. Pattern variables in code patterns are indicated by escaping variables in the code pattern.

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

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

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

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

We use higher-order pattern variables[22, 21] for code patterns that contain binding occurrences, such as lambda expressions, let expressions, do expressions, or functions.

For example, a high-order pattern that matches the code of a function `<fn x => ...>`, of type `<'a -> 'b>` is written in eta-expanded form `<fn x => ~(g <x>)>`. When the pattern matches, the matching binds the higher-order pattern variable `g` to a function with type `<'a> -> <'b>`

Every higher order pattern variable must be in fully saturated form, by applying it to all the bound variables of the code pattern. For example if `g` is a higher-order pattern variable with type `<'a> -> <'b> -> <'c>` then we must write `~(g <x> <y>)`. The arguments to the higher-order pattern variable must be explicit bracketed variables, one for each variable bound in the code pattern at the context where the higher-order pattern appears. A higher-order pattern variable is used like a function on the right-hand side of a matching construct.

For example functions which implement the three monad axioms are written as follows:

```
fun monad1
    <do mswo
        { x <- return mswo ~e
        ; ~(z <x>) }>
= z e

fun monad2 <do mswo { x <- ~m; return x }> = m
```

```
fun monad3
    <do mswo
        { x <- do mswo {y <- ~a
        ; ~(b <y>)}
        ; ~(c <x> }>
= <do mswo { y' <- ~a
           ; do mswo { z <- ~(b <y'>)
           ; ~(c <z>) }}>
```

When the function `monad1` is applied to the code `<do mswo {a <- returm mswo (g 3); h(a + 2)}>`, the pattern variable `e` is bound to the function `fn x => <h(~x + 2)>` which has the type `<int> -> <int M>`. The right-hand side of `monad1` rebuilds a new code fragment, substituting formal parameter `x` of `e` by `<g 3>`, constructing the code `<h((g 3)+ 2)>`.

This technique can be used to build optimizations, or to translate a residual program into a target language.

7 Conclusion

The important issues of efficient language implementation by refinement from high-level specifications are: the efficient use of the underlying target environment, and removing the layer of interpretative computation introduced by such specifications. We have shown that monads and staging are the right abstraction mechanisms to accomplish the task. To effectively use these tools we propose that DSL implementers follow a well defined method. We reiterate our method here:

- **Domain analysis.** The problem domain is analyzed to find the common abstractions around which the language is designed. This step is perhaps the most important step in a good language design. It has been studied extensively by others [32, 2, 3]. Our research group has been investigating the integration of DSL design and domain analysis for several years. Recently Widen and Hook have summarized a "top level" view of this integration, which is called the Software Design Automation (SDA) method [33]. This method provides a design process and many synthesis techniques to facilitate the integration of traditional domain analysis activities with language design and implementation. The method we propose can be used in the context of SDA. It specifically addresses the language implementation phase of the process.
- **Definitional interpreter.** Once the language has been identified, the next step is to provide it with a semantics given as a pure functional interpreter. This program can be thought of as its high-level definition [14, 25]. high-level interpreters are usually easy to construct and provide a reference which can be consulted to resolve any ambiguity in the language specification discovered in further steps. By building it in an executable framework (a functional language, such as Haskell or ML) it also provides a rapid prototype against which expectations can be measured.
- **Binding time improvements.** The next step requires a binding separation [8]. By identifying compile-time versus run-time data structures in the definitional interpreter, we can separate those with *both* components into separate data-structures. Examples of binding time improvements include the separation of environments, which map names to values, into a compile-time index and a run-time stack, and the introduction of a local recursive function to separate the recursion which drives the analysis of the syntax of the program being interpreted from the recursion that encodes the looping of the `while` command.
- **Target domain analysis**. The next step is to analyze the target language to identify the primitive implementation features that will support the translation. This step is usually straight-forward as the target language is often fixed, and well understood.
- **Design a monad.** The next step is to design a monad to capture the effects and actions implicit in the target language. This is a hard step in the process since it requires both abstract knowledge about the structure and properties of monads, and detailed concrete knowledge about the target domain. The choices made in this step influence the structure of the monad, the structure of the monadic interpreter, and the run-time system which interacts with the low-level effects of the target language.

  Once the monad is designed, an implementation for the monad as a pure functional emulation must be produced. The implementation must emulate the actions in a purely functional setting by explicitly threading abstract representations of the actions such as "stores", "I/O streams", or "exception continuations" in and out of all computations.

- **Monadic Interpreter.** The next step is to refine the purely functional definitional interpreter into one written in a monadic style [28, 24, 13]. This implementation is still purely functional because the actions of the monad are emulated in a functional style. But because the actions are now explicit, we have moved the form of definition closer to the target language. This step often requires a big change to the structure of the source code, because the monad makes implicit much of the "plumbing" explicit in the interpreter. The cost of this restructuring is not without benefit. The removal of the explicit plumbing results in programs which are simpler, and more immune to future changes.

- **Staging.** The next step completes the binding-time separation begun in the binding time improvement step. That step separated the compile-time *data* from the run-time data. Staging separates the compile-time *computations* from the run-time computations. This is done by placing explicit staging annotations in the program written in MetaML. Staging is the crucial step that differentiates an (inefficient) interpreter from an (efficient) compiler.

- **Transformation of residual code.**

  The residual object-program produced by a staged interpreter is both a data structure that can be manipulated, and a program that can be run. Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. The restricted form of the residual object program make it possible to use the intensional analysis of object-code tools provided by MetaML to easily build the final translation step to the target language.

7.1 Benefits of the approach

This paper illustrated a step by step method for constructing correct and efficient implementations of DSLs. The method has the following advantages over building a DSL implementation in an ad-hoc fashion.

- **Simplicity.** We divide the task of DSL implementation of DSL into small manageable tasks. The compiler is constructed by a method of refinement, and we use special abstraction mechanisms so that each step addresses only a single aspect of the compiler.

- **Reuse.** Our method provides many opportunities for reuse. By using the abstraction methods of monads and staging, much of the code remains unchanged between refinement steps. In addition, monad implementations are reusable across DSLs, and multiple DLS using the same target language can reuse the intensional analysis.

- **Control.** Instead of using a fixed set of techniques or tool to generate compilers, we outline a method which provides users control over each step. A good impedance match between low-level features of the target language and the high-level DSL is necessary for good performance. Since every compiler is different, users need such fine grained control.

- **Correctness.** The MetaML type system provides major support for ensuring the correctness of the compilers generated. It is simply not possible to write a type-incorrect translation. But type-correctness is not enough. We wish to prove other correctness properties as well, such as the equivalence between the artifacts produced by each step of the method. We believe that it is possible for each step to make explicit its proof obligations, and because each step produces a functional program, it is possible to use equational reasoning to prove these obligations

7.2 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 laws so compelling that we implemented them in an ad-hoc fashion inside the MetaML system.

8 Acknowledgments

The authors would like to acknowedge generous support from the USAF Air Materiel Command, contract #F19628-96-C-0161; the National Scince Foundation, grants #IRI-9625462 and #CCR-9803880; and the Department of Defense.

References

[1] 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.

[2] Grady Campbell. Abstraction-based reuse repositories. Technical Report REUSE-REPOSITORIES-89041-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, June 1989.

[3] Grady Campbell, Stuart Faulk, and David Weiss. Introduction to Synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, 1990.

[4] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The 23$^{\rm rd}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.

[5] O Danvy, J Koslowski, and K Malmkjaer. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 91.

[6] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[7] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *In FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 52–64. ACM Press, 1993. (Appears, in extended form, in the Journal of Functional Programming, 5, 1, Cambridge University Press, January 1995.).

[8] 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).

[9] 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.

[10] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.

[11] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[12] 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, Linkoping, Sweden, January 1996.

[13] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[14] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Com-*

puter Science. Elsevier Science Publishers B. V. (North-Holland), 1990.

[15] Peter D. Mosses. SIS-semantics implementation system, reference manual and users guide. Technical Report DAIMI report MD-30, University of Aarhus, Aarhus, Denmark, 1979.

[16] Peter D. Mosses. Action semantics. *Cambridge Tracts in Theoretical Computer Science*, (26), 1992.

[17] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[18] L. Paulson. *Methods and Tools for Compiler Construction, B. Lorho (editor)*. Cambridge University Press, 1984.

[19] Lawrence Paulson. A sematics directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–233. ACM, January 1982.

[20] 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.

[21] Frank Pfenning, Jolle Despeyroux, and Carsten Schrmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997.

[22] Frank Pfenning, Gilles Dowek, Thrse Hardin, and Claude Kirchner. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, Bonn, Germany, September 1996.

[23] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[24] Guy Steele. Building interpreters by composing monads. In *21st Annual ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.

[25] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.

[26] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, Aalborg, Denmark, 13–17July 1998.

[27] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

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

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

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

[31] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[32] Tanya Widen. Formal language design in the context of domain engineering. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1997.

[33] Tanya Widen and James Hook. Software design automation: Language design in the context of domain engenieering. In *The 10th International Conference on Software Engineering & Knowledge Engineering (SEKE'98)*, pages 308–317, San Francisco Bay, California, June 1998.