Modular Domain Specific Languages and Tools

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520
paul.hudak@yale.edu

Abstract

A domain specific language *(DSL) allows one to develop software for a particular application domain quickly and effectively, yielding programs that are easy to understand, reason about, and maintain. On the other hand, there may be a significant overhead in creating the infrastructure needed to support a DSL. To solve this problem, a methodology is described for building domain specific* embedded *languages (DSELs), in which a DSL is designed within an existing, higher-order and typed, programming language such as Haskell or ML. In addition, techniques are described for building modular interpreters and tools for DSELs. The resulting methodology facilitates reuse of syntax, semantics, implementation code, software tools, as well as look-and-feel.*

Keywords: *software reuse, modularity, abstraction, domain specific languages, functional languages, formal methods.*

1 Introduction

A *domain specific language* (DSL) is a programming language tailored for a particular application domain. Characteristic of an effective DSL is the ability to develop complete application programs for a domain quickly and effectively. A DSL is not (necessarily) "general purpose." Rather, it should capture precisely the semantics of an application domain, no more and no less. Bentley also makes a strong argument for DSLs as "little languages" [Ben86].

Common examples of DSLs include Lexx and Yacc for lexing and parsing programs, PERL for text manipulation, VHDL for hardware description, TeX and LaTex for document preparation, HTML and SGML for document markup, Tcl/Tk for GUI scripting, VRML and Open GL for 3D graphics, Mathematica and Maple for symbolic computation, and AutoLisp and AutoCAD for computer-aided design. Some purported general-purpose languages can also be said to be domain specific. For example, Prolog is excellent for applications specified using predicate calculus, and functional languages such as Haskell and ML for functional specifications (Haskell is sometimes referred to as a DSL for denotational semantics).

There are lots of advantages to using DSLs, starting with the fact that programs are generally easier to write, reason about, and modify compared to equivalent programs written in general purpose languages. Indeed, these are the same advantages gained from using any high-level programming language. Arguably, a good DSL is at an even higher level than a conventional high-level language, and can often be used by those who are not expert programmers. Mathematicians can easily learn Mathematica, paper writers can learn Latex, hardware designers can learn VHDL, and so on. In sophisticated domains, the *domain engineer* is the person we want to use a DSL.

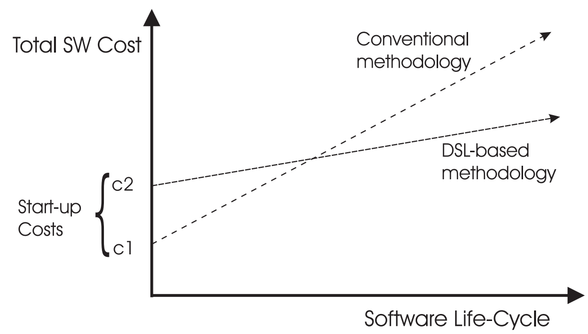

Figure 1: The Payoff of DSL Technology

A very rough (and admittedly over-simplified) quantitative argument in favor of using DSLs for software development is illustrated in Figure 1. The point is, the initial cost of DSL development may be high compared to the equivalent cost of "tooling up" for an application under a more traditional software development scenario. But the slope of the curve for aggregate software development cost should be considerably lower using a DSL, and thus at some point the DSL approach should yield significant savings.

2 The Problem

Unfortunately, it can be fairly difficult to design and implement a programming language from scratch. Moreover, there's a good chance that we won't get it right the first time; it will evolve, and we will experience all of the difficulties associated with that evolution. In other words, what if the start-up costs shown in Figure 1 are so high that we never break even? Or what if we get it all wrong, and incur the start-up cost several more times during a software system's life-cycle? In other words, is in fact the DSL approach really practical?

In this paper I outline several techniques that I believe can lead to the effective use of this methodology. These techniques rest on two key thoughts:

1. We begin with the assumption that we really don't want to build a new programming language from scratch. Better, let's inherit the infrastructure of some other language—tailoring it in special ways to the domain of interest—thus yielding a domain-specific *embedded* language (DSEL).

2. Building on this base, we can then concentrate on semantic issues. Sound abstraction principles can be used at this level to build language tools that are themselves easy to understand, highly modular, and straightforward to evolve.

With this approach one can create a rich infrastructure that facilitates reuse of syntax, semantics, implementation code, software tools, look-and-feel, and various other related artifacts. With such an infrastructure, the savings implied by Figure 1 are more likely to be achievable.

The idea of embedding a DSL within an existing language is not new, of course. Lisp macros have been used for years to develop embedded languages. Modern object-oriented approaches such as Jakarta [BLS98] take a software-generator approach to the problem: a DSL is a specification language that a software generator uses to create the program of interest. Our approach, however, is distinctive in two ways. First, it is based on a *pure* embedding—no pre-processor, macro-expander, or generator. Second, it emphasizes the importance of *semantics*, as manifested through modular algebraic approaches to DSL implementation.

In the remainder of this paper I describe the results of using the functional language Haskell [HPJWe92] to build DSELs. Haskell has several features that I believe are crucial to the pure embedding of a DSL: higher-order functions, lazy evaluation, polymorphism, and type classes. A language with less of these features could possibly also be used, but probably only with a higher emphasis on pre-processing and the complexities thus introduced.

3 Syntax vs. Semantics

Tools such as Lex [Les75] and Yacc [Joh75], as well as more sophisticated programming environment generators (e.g. [Rep84]), have been shown to be quite useful in designing new programming languages; they are certainly better than building lexers, parsers, and other tools from scratch. On the other hand, syntactic minutiae should arguably be the least of a language designer's worries. This is another twist on the slogan "semantics is more important than syntax" often bellowed in programming language circles. This is not to say that syntax does not matter—I believe that it does—but rather places syntax in proper perspective.

However, even when one focuses on semantic issues, many of the details still do not matter much. Examples of semantic minutiae include numbers, booleans, and other simple datatypes and their operators; scoping rules; looping constructs; pattern-matching rules; endless details in the type system and module system; etc. Of course there are many deeper semantic issues—such as the evaluation order of arguments, and higher-order constructs and values—but in most domains there are many reasonable choices in the spectrum of possibilities of these features.

3.1 DSELs Inherit Language Features

So the point is, instead of designing a programming language from scratch, why not borrow most of the design decisions made for some other language? And while we're at it, let's borrow as many as we can of the tools designed for this other language as well. We call this a domain-specific *embedded* language, or DSEL. Aside from the obvious advantage of being able to reuse many ideas and artifacts, DSELs have certain other advantages over DSLs:

First off, although I pointed out earlier that a DSL "should capture precisely the semantics of an application domain, no more and no less," a DSL in fact is often not used in total isolation. Users of even (or perhaps especially) the most elegant DSLs may find themselves frustrated at not having access to more general programming language features. Indeed, a common evolutionary path of DSL design is to begin with modest goals—usually achieved quickly—and to end with a complex general purpose language—usually achieved after much time and effort—where one has to look hard to find the pure domain-specific abstractions that were its foundation.

Secondly, if we design several DSELs for different domains, all derived from the same base language, then programmers in the different domains can share a common core language, along with all its associated tools. Indeed, in a large application it is quite conceivable to have more than one DSEL. For them all to have a similar look-and-feel is a clear advantage.

3.2 An Example

It is surprisingly straightforward to design a DSEL for many specific applications. We and others in the Haskell community have done so using Haskell in a variety of domains: parser generation, graphics, animation, simulation, music composition, hardware design, VLSI layout, pretty printing, concurrency, GUIs, component scripting [PJML98], and geometric region analysis, to name a few. Each of these applications was a *pure embedding*: neither Haskell semantics nor implementation was modified, nor was a pre-processor used to add extra language features. Everything was written entirely in standard Haskell.

As an example, a DSEL for the domain of geometric region analysis came about through an experiment conducted jointly by Darpa, ONR, and the Naval Surface Warfare Center. This well-documented experiment (see [Car93, CHJ93, LBK+94]) demonstrates not only the viability of the DSEL approach, but also its evolvability. Three different versions of the system were developed, each capturing more advanced notions of the target system, with no *a priori* knowledge of the changes that would be required. The modularity afforded by the DSEL made these non-trivial changes

```
-- Geometric regions are represented as functions:
  type Region  =  Point -> Bool

  -- So, to test a point's membership in a region:
  inRegion       :: Point -> Region -> Bool
  p `inRegion` r = r p

  -- Given suitable definitions of "circle",
  -- "outside", and (/\):
  circle    :: Radius -> Region
            -- creates a region with given radius
  outside   :: Region -> Region
            -- the logical negation of a region
  (/\)      :: Region -> Region -> Region
            -- the intersection of two regions
  (\/)      :: Region -> Region -> Region
            -- the union of two regions

  -- We can then define an annulus:
  annulus      :: Radius -> Radius -> Region
  annulus r1 r2 = outside (circle r1) /\ (circle r2)
```

Figure 2: Example of a DSEL for a Naval Application

quite easy to incorporate.

The resulting notation was not only easy to design, it was also easy to use and reason about. Figure 2 shows some of the code to give the reader a feel for its simplicity and clarity.[1] Because the domain semantics is captured concisely, it is possible even for non-programmers to understand much of the code. In the NSWC experiment, those completely unfamiliar with Haskell were able to grasp the concepts immediately. Some even expressed disbelief that the code was actually executable.

(Indeed, despite the presence of this last sentence, one reviewer of the first draft of this paper complained that "the paper claims to be interested in both syntax and semantics, [but] the presented details are mostly syntactic (e.g., the definition `inRegion`), and the paper makes no attempt to distinguish mathematical and programmatic entities." But in fact this definition of `inRegion` is entirely semantic. Furthermore, equational reasoning, as described below, allows one to blur the distinction between mathematical and programmatic entities: programs can be viewed as specifications. This is a feature, as it enhances the application of formal methods.)

Note that operators such as `(/\)`, `(\/)` and `outside` take regions as arguments. But regions are themselves represented as functions, so it it not surprising that higher-order functions are the key underlying abstraction needed to create this simple DSL. For example, the definition of `(/\)` is given by:

```
(r1 /\ r2) p = r1 p && r2 p
```

which is equivalent to:

```
p `inRegion` (r1 /\ r2) =
  (p `inRegion` r1) && (p `inRegion` r2)
```

and which can be read quite naturally as: "a point `p` lies in the intersection of `r1` and `r2` if it lies in both `r1` and `r2`."

Another important advantage of the DSEL approach is that it is highly amenable to formal methods, especially when using a language such as Haskell with a simple underlying semantics. The key point is that, once a set of axioms is established, one can reason directly *within the domain semantics*, rather than within the semantics of the programming language. In the NSWC experiment we straightforwardly proved several axioms of our DSEL that would have been much more difficult to prove in most of the competing designs. As a simple example, to prove associativity of region intersection:

```
(r1 /\ r2) /\ r3 = r1 /\ (r2 /\ r3)
```

we can use the definition of `(/\)` given above to reason equationally:

```
((r1 /\ r2) /\ r3) p
= (r1 /\ r2) p && r3 p
= (r1 p && r2 p) && r3 p
= r1 p && (r2 p && r3 p)
= r1 p && (r2 /\ r3) p
= (r1 /\ (r2 /\ r3)) p
```

Indeed, what often arises out of this use of formal methods is a rich *algebra* that captures the domain semantics quite nicely. This is elaborated on in the next section.

4 Modular Algebraic Semantics

In a later section I describe how an implementation of a DSL can be constructed in a modular way, thus facilitating reuse of software components across possibly many DSL design efforts. The root of that process, however, is a good understanding of the domain semantics itself: one that recognizes layers of abstraction rather than one monolithic structure.

4.1 Simple Graphics

To demonstrate this, let's look at a simplified version of *Fran* [EH97, Ell97], a DSEL that we have developed in collaboration with Microsoft, for "functional reactive animation." We begin with some simple operators for manipulating graphical objects, or "pictures," as shown in Figure 3.[2] With these operators a rich algebra of pictures can be established. For example, `scale`, `color`, and `trans` all distribute over `over`, `above`, and `beside`, and the latter three are all associative. With these axioms many useful properties of graphical objects can be established.

[1] In Haskell, any function can be used in infix style by enclosing it in back-quotes. Thus `` p `inRegion` r `` is the same as `inRegion p r`.

[2] These are not unlike those for geometric regions given previously, but are even more like Henderson's functional graphics given in [Hen82].

```
-- Atomic objects:
circle          -- a unit circle
square          -- a unit square
import "p.gif"  -- an imported bit-map

-- Composite objects:
scale    v p   -- scale picture p by vector v
color    c p   -- color picture p with color c
trans    v p   -- translate picture p by vector v
p1 `over`   p2  -- overlay p1 on p2
p1 `above`  p2  -- place p1 above  p2
p1 `beside` p2  -- place p1 beside p2
```

Figure 3: A Simple Graphics DSEL

4.2 Simple Animations

Next, we note a very simple relationship between pictures and animations: an animation is simply a time-varying picture! In Haskell we can express the type signature for animations by writing:

```
type Animation = Time -> Picture
```

which means that an animation is a function from time to pictures. But in fact *many* sorts of things can be time varying. Thus we adopt a more generic viewpoint by defining the notion of a (polymorphic) *behavior*, and then defining animations in terms of it:

```
type Behavior a = Time -> a
type Animation  = Behavior Picture
```

Now for the key step, we can "lift" all of our operators on pictures to work on behaviors as well. For example:[3]

```
(b1 `overB`   b2) t = b1 t `over`   b2 t
(b1 `aboveB`  b2) t = b1 t `above`  b2 t
(b1 `besideB` b2) t = b1 t `beside` b2 t
```

We can also lift the other operators, keeping in mind that the vector and color arguments themselves might be time varying, and so we write:

```
(scaleB v b) t    = scale (v t) (b t)
(colorB c b) t    = color (c t) (b t)
(transB v b) t    = trans (v t) (b t)
```

Indeed, using higher-order functions we can write functions to lift any other function of a given arity. For example, for arities zero, one, and two we can define:

```
(lift0 v)       t = v
(lift1 f b1)    t = f (b1 t)
(lift2 f b1 b2) t = f (b1 t) (b2 t)
```

[3]This lifting can be done more elegantly by using Haskell's *type classes* to overload the operators `over`, `above`, etc., but for simplicity this technicality is avoided in this paper; see [EH97] for details.

Now the previous functions can be defined more simply by:

```
scaleB = lift2 scale
colorB = lift2 color
transB = lift2 trans
```

Lots of new functions can be defined as well, such as:

```
sinB = lift1 sin
cosB = lift1 cos
```

The function `lift0` is used to lift constants to constant functions. For example, `lift 3.14159` is equivalent to the constant function:

```
pi t = 3.14159
```

Indeed, using Haskell type classes, every literal constant can be automatically lifted. For example, the literal 42 is equivalent to the constant function `b42` defined by:

```
b42 t = 42
```

Finally, we define a behavior that reflects the current time:

```
time t = t
```

Higher-order functions obviously play a crucial role in this process of "lifting" values from one level of functionality to another. Living in a world where everything is lifted is actually fairly natural. For example, with the above liftings we can now express continuous-time animations. Let's first define a couple of simple utility behaviors. The first is a numeric behavior that varies smoothly and cyclically between -1 and +1:

```
wiggle = sinB (pi*time)
```

where we assume that `*` is also lifted. Using `wiggle` we can then define a function that smoothly varies between its two argument values.

```
wiggleRange lo hi = lo + (hi-lo) * (wiggle+1)/2
```

where 1 and 2 are lifted literals. Finally, let's create a very simple animation: a red, pulsating ball.

```
ball = colorB red
         (scaleB (wiggleRange 0.5 1) circle)
```

This is an extremely concise animation program. The equivalent program in Java, for example, is dozens of lines long.

We can also develop a rich *algebra of animation*. In fact, *the entire algebra of pictures generalizes directly to animations*. And with time as a first-class value, there are even more opportunities for expressiveness if we add time-specific operators. For example, in Fran we have an operator for expressing *time transformations*, and thus:

```
anim `aboveB` (timeTransform (-1) anim)
```

displays two copies of the animation `anim`, one just above the other and delayed by 1 second.

Perhaps more importantly, Fran has an operator for expressing *integration* over time. To express the behavior of a falling ball, for example, we can write:

```
let y = y0 + integral v
    v = v0 + integral g
in translate (x0,y) ball
```

where `(x0,y0)` is the initial position of the ball, `v0` it its initial velocity, and `g` is gravity. These equations can be read literally as the standard equations learned in introductory physics to describe the same phenomenon. Indeed, partial differential equations in general can be written and directly executed in Fran.

As you might guess, we can also develop a useful *algebra of time*, which includes such basic axioms as:

```
timeTransform f (timeTransform g b)
                     = timeTransform (f.g) b
integral k           = k*time
integral time        = 0.5*time**2
integral (sin time) = cos time
```

where `(f.g)` denotes the composition of the functions (behaviors) `f` and `g`.

4.3 Reactive Animations

For the third and final layer of our semantic structure, we add *reactivity*. This layer is reminiscent of CSP or similar process algebra, and is based on a notion of an *event*. Primitive events include things like mouse clicks and key presses, but additionally include *predicate events* such as `time>5`. There are also ways to combine events and filter them. The basic reactive expression has the form:

```
b1 `until` e => b2
```

which can be read: "behave as `b1` until event `e` occurs, then behave as `b2`." Despite what looks like special syntax, we emphasize again that this is a pure embedding in Haskell: `until` and `(=>)` are just functions, and the above fragment is equivalent to:

```
until b1 (e => b2)
```

where `=>` is an infix operator like `+` or `*`.

As a simple example of *recursive* reactivity, here is a circle that changes color everytime the left mouse-button is pressed:

```
color (cycle red green blue) circle
  where cycle c1 c2 c3 =
          c1 `until` lbp => cycle c2 c3 c1
```

`lbp` is the event associated with a left mouse-button press. Note how the recursive call to `cycle` permutes the color arguments, thus causing the behavior to change color everytime the left mouse-button is pressed.

This example highlights the utility of lazy evaluation in pure embeddings of DSLs. In particular, if the functions `until` and `(=>)` were strict, the above progam would not terminate. A similar functionality could be encoded in a strict language such as ML, but such encodings are generally cumbersome and less natural.

It turns out the previously described algebra of animations still holds in the reactive framework—nothing "gets broken"—and additionally there is an algebra of reactivity that is reminiscent of that for other process calculii. (Further details on the design, semantics, and implementation of reactivity is beyond the scope of this paper, but may be found in [EH97, Ell97].)

5 Modular Monadic Interpreters

A DSEL in Haskell can be thought of as a higher-order algebraic structure, a first-class value that has the "look and feel" of special syntax. In some sense it is still just notation; its semantics is captured by an *interpreter*. For example, although the program for the red pulsating ball can be thought of as executing on its own, it is better to think of the existence of an interpreter to give it meaning, and the software that implements it can be structured accordingly. This permits a key opportunity for modular design, in turn facilitating reuse of the interpreter building blocks, and evolution of the system since changes in the domain semantics are in many cases inevitable.

The design of truly modular interpreters has been an elusive goal in the programming language community for many years. In particular, one would like to design the interpreter so that different language features can be isolated and given individualized interpretations in a "building block" manner. These building blocks can then be assembled to yield languages that have only a few, a majority, or even all of the individual language features. Progress by Moggi, Espanol, and Steele [Mog89, Ste94, Esp93, Esp95] laid the groundwork for our recent effort at producing a modular interpreter for a non-trivial language [LHJ95], and basing modular compiler construction technology on it [LH96, Lia98]. The use of *monads* [PJW93, Wad90] to structure the design was critical.

Our approach means that language features can be added long after the initial design, *even if they involve fundamental changes in the interpreter functionality.* For example, we have built a series of languages and interpreters that begin with a small calculator language (just numbers), then a simple first-order language with variables, then a higher-order language with several calling conventions, then a language with errors and exceptions, and so on, as suggested in Figure 4. At each level the new language features can be added, along with their semantics, *without altering any previous code.*

It is also possible with this approach to capture not only domain-specific semantics, but also domain-specific *optimizations*. These optimizations can be done incrementally and independently from each other and from the core semantics. We have used this to implement traditional compiler optimizations [LH96, Lia98], but the same techniques could be used for domain-specific optimizations.

To get a feel for how a monadic interpreter works, note that a conventional interpreter maps, say, a term,

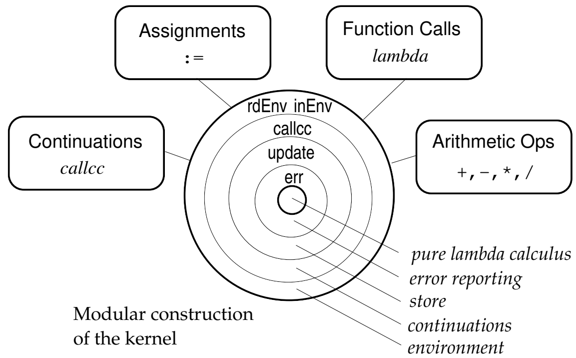

Figure 4: Modular monadic interpreter structure

environment, and store, to an answer. In contrast, a monadic interpreter maps terms to *computations*, where the details of the environment, store, etc. are "hidden" in the computation. Specifically:

```
interp :: Term -> InterpM Value
```

where `InterpM Value` is the computation monad of final answers.

What makes the interpreter modular is that all three components above—the term type, the value type, and the monad—are *configurable*. To illustrate, if we initially wish to have an interpreter for a small number-expression language, we can fill in the definitions as follows:

```
type Value   = OR Int Bottom
type Term    = TermA
type InterpM = ErrorT Id
```

The first line declares the answer domain to be the union of integers and bottom. The second line defines terms as `TermA`, the abstract syntax for arithmetic operations. The final line defines the interpreter monad as a transformation of the identify monad `Id`. The monad transformer `ErrorT` accounts for the possibility of errors; in this case, arithmetic exceptions. At this point the interpreter behaves like a calculator:

```
Run> ((1+4)*8)
     40
Run> (3/0)
     ERROR: divide by 0
```

Now if we wish to add function calls, we can extend the value domain with function types, add the abstract syntax for function calls to the term type, and apply the monad transformer `EnvT Env` to introduce an environment `Env`.

```
type Value   = OR Int (OR Function Bottom)
type Term    = OR TermF TermA
type InterpM = EnvT Env (ErrorT Id)
```

Here is a test run:

```
Run> ((\x.(x+4)) 7)
     11
Run> (x+4)
     ERROR: unbound variable: x
```

We can further add other features—such as conditionals, lazy evaluation, letrec declarations, nondeterminism, continuations, references, and assignment—to our interpreter, as suggested in Figure 4. Whenever a new value domain (such as Boolean) is needed, we extend the `Value` type; and to add a new semantic feature (such as a store or continuation), we apply the corresponding monad transformer.

5.1 Language Tools and Instrumentation

It is also possible to add "non-standard" features to a programming language, such as debugging, tracing, profiling, performance monitoring, etc. Although these feature may be non-standard in a technical sense, they are vitally important to effective software development, including any methodology that is using a DSL. A disciplined approach to designing such tools will surely benefit the software development process. Our framework for modular interpreters can in fact handle these non-standard features straightforwardly.

The advantage of a modular approach to language tool construction is that tools can be layered onto the system without affecting each other; changes and additions are thus easily accomplished. A tool building block specified in our framework can be automatically combined with the corresponding standard semantics building block to yield a composite semantics that incorporates the behaviors of both. This also means that a tool building block—say a profiler—may be used for different language or DSL implementations—say Fran and geometric region analysis. The opportunities for code reuse are thus enormous. Figure 5 shows the compositional nature of this methodology, and Figure 6 shows a flow diagram.

For an example of these ideas in action, consider this simple factorial program written in a hypothetical DSL:

```
fact =
  "let mul(x,y) = {Profile mul}:(x*y)
   in let fac(n,acc) =
            {Profile fac}:
              if n==0 then acc
              else fac(n-1, mul(n,acc))
      in fac 3 1"
```

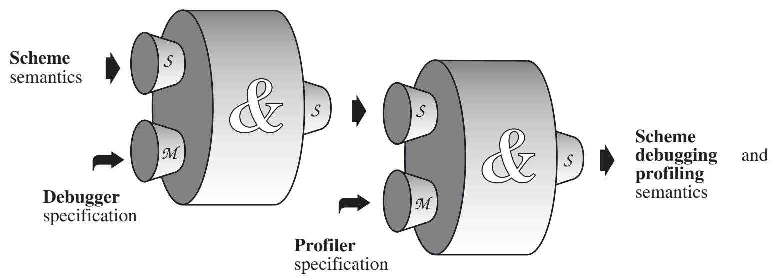

Figure 5: Composing monitors

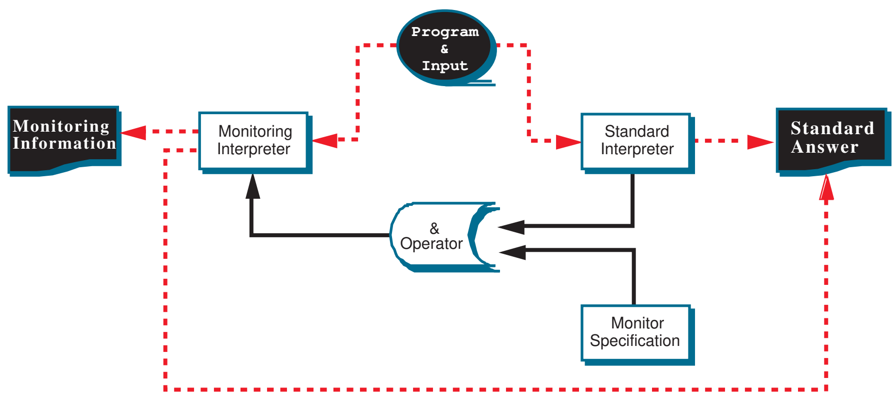

Figure 6: System diagram

The occurrences of `Profile` are annotations which are treated as language extensions, and their meaning is captured precisely in a profiler building block. Following the framework implied by Figures 5 and 6, we can test the profiler on the above program using both a lazy and eager interpreter:

```
Run> execute (profiler & eager) fact
(6, [(fac,4), (mul,3)])

Run> execute (profiler & lazy) fact
(6, [(fac,4), (mul,3)])
```

In this example the profiling results for both interpreters are the same. However, this is not always the case; for example, if we change the consequent branch in `fact` to 1 rather than `acc` (a plausible error):

```
badFact =
  "let mul(x,y) = {Profile mul}:(x*y)
   in let fac(n,acc) =
             {Profile fac}:
               if n==0 then 1
               else fac(n-1, mul(n,acc))
      in fac 3 1"
```

then the lazy profiler result differs from the eager one because `acc` is never demanded by the lazy interpreter:

```
Run> execute (profiler & eager) badFact
(1, [(fac,4), (mul,3)])

Run> execute (profiler & lazy) badFact
(1, [(fac,4)])
```

Using these basic ideas, rather sophisticated debuggers for a variety of languages can be quickly developed [KH95, Kis92].

6 Partial Evaluation

Perhaps all of this seems too good to be true. Indeed, there is one major drawback to our approach to modular interpreter construction: each building block imposes an independent layer of interpretive overhead, resulting in seemingly impractical interpreters for any realistic DSL. Although our modular monadic approach can be used to reason about compiler construction [LH96], we would prefer to use (and reuse) the modular interpreters.

The solution is to use *partial evaluation*. In particular, we can use partial evaluation to optimize the composed interpreters described earlier in two ways: (1) specializing each language feature building block (which may be a non-standard tool-oriented feature) with respect to the one below it, yielding a *concrete interpreter*; and (2) specializing the concrete interpreter (from the previous step) with respect to a source program, yielding an *instrumented program*; i.e. a program with embedded code to perform, for example, debugging actions. Figure 7 provides a useful viewpoint of these two levels of optimization. Similar results as these can be achieved in object-oriented systems using design patterns [GHJV95] and more directed ap-

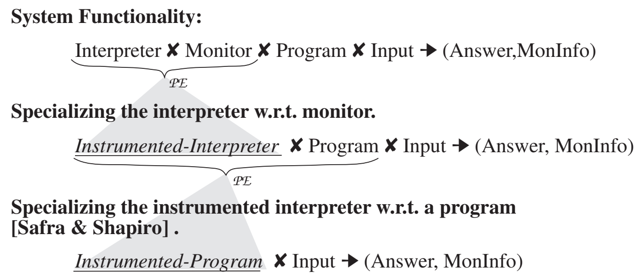

Figure 7: Partial evaluation optimization levels

Program	*Unoptimized system (ms)*	*Instrumented interpreter (ms)*	*Instrumented program (ms)*	*Total speedup*
fac	478.42	11.20 (×43)	0.69 (×16)	×693
power2	568.17	14.17 (×40)	0.34 (×42)	×1671
deriv	2642.00	61.53 (×40)	0.88 (×70)	×2797
qsort	1554.50	36.82 (×42)	2.34 (×16)	×664
nsqrt	494.00	12.08 (×41)	1.16 (×10)	×425

Figure 8: Improvements Due to Partial Evaluation

proaches to tool generation [DRW96].

We have used existing partial evaluation techniques to do this, with dramatic improvements in performance. Unfortunately, there does not currently exist a suitable, easy-to-use partial evaluator for Haskell. Our approach was to convert the Haskell program to Scheme, partially evaluate the Scheme program, and then translate back into Haskell. This is not a fully automated process, and the lack of a good partial evaluator for Haskell remains as the one stumbling block to more effective use of our overall methodology.

In any case, Figure 8 compares the speedups gained by partial evaluation for some benchmark programs. The table shows the execution times for the unoptimized system, the instrumented interpreter, and the instrumented program. Each optimization removes one level of interpretation which results in the speedups shown in parentheses. Every interpretation level contributes a slowdown of about 15-70 times. By removing these levels of interpretation using partial evaluation, the speedup gained is up to three orders of magnitude (the largest speedup being 2797). These results dramatically reveal the advantage (and importance!) of partial evaluation.

7 Conclusion

I have described a methodology for designing and implementing domain-specific embedded languages. This collection of techniques has never been collected together and presented as a single unified methodology before. It is especially targeted for the software reuse community, which is not likely to be familiar with many of these ideas, and for which the methodology offers a high degree of reuse: of syntax, semantics, implementation code, software tools, look-and-feel, and related artifacts. Except for the lack of an effective partial evaluator for Haskell, all of these techniques can and are being used to create DSELs in a variety of applications.

Acknowledgments

Thanks to the Yale Haskell Group for contributing so much to the ideas in this paper. This research was supported in part by DARPA under grant number F30602-96-2-0232, and NSF under grant number CCR-9633390.

References

[Ben86] Jon Bentley. Little languages. *CACM*, 29(8):711–721, 1986.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: A tool suite for building GenVoca generators. In *Proceedings of 5th International Conference on Software Reuse*. IEEE/ACM, 1998.

[Car93] J. Caruso. Prototyping demonstration problem for the prototech hiper-d joint prototyping demonstration project. CCB

Report 0.2, Naval Surface Warfare Center, August 1993.

[CHJ93] W.E. Carlson, P. Hudak, and M.P. Jones. An experiment using Haskell to prototype "geometric region servers" for navy command and control. Research Report 1031, Department of Computer Science, Yale University, November 1993.

[DRW96] P. Devanbu, D. Rosenblum, and A. Wolf. Generating testing and analysis tools. *ACM Transactions on Software Engineering and Methodology*, 1996.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[Ell97] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*. USENIX, October 1997.

[Esp93] David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.

[Esp95] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattens: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*, pages 179–187. ACM, 1982.

[HPJWe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Nonstrict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[Joh75] S.C. Johnson. Yacc – yet another compiler compiler. Technical Report 32, Bell Labs, 1975.

[KH95] A. Kishon and P. Hudak. Semantics-directed program execution monitoring. *Journal of Functional Programming*, 5(4), October 1995.

[Kis92] A. Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring*. PhD thesis, Yale University, Department of Computer Science, 1992.

[LBK+94] J.A.N. Lee, B. Blum, P. Kanellakis, H. Crisp, and J.A. Caruso. ProtoTech HiPer-D Joint Prototyping Demonstration Project, February 1994. Unpublished; 400 pages.

[Les75] M.E. Lesk. Lex – a lexical analyzer generator. Technical Report 39, Bell Labs, 1975.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *European Symposium on Programming*, April 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of 22nd ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, January 1995. ACM Press.

[Lia98] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, Department of Computer Science, May 1998.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[PJML98] Simon Peyton-Jones, Erik Meijer, and Dan Leijen. Scripting COM components in haskell. In *Proceedings of 5th International Conference on Software Reuse*. IEEE/ACM, 1998.

[PJW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993. (to appear).

[Rep84] T. W. Reps. *Generating Language-Based Environments*. The MIT Press, 1984.

[Ste94] Guy L. Steele Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 472–492, New York, January 1994. ACM Press.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.