Domain Specific Languages[*]

Paul Hudak
Department of Computer Science
Yale University

December 15, 1997

1 Introduction

When most people think of a programming language they think of a *general purpose* language: one capable of programming any application with relatively the same degree of expressiveness and efficiency. For many applications, however, there are more natural ways to express the solution to a problem than those afforded by general purpose programming languages. As a result, researchers and practitioners in recent years have developed many different *domain specific* languages, or DSL's, which are tailored to particular application domains. With an appropriate DSL, one can develop complete application programs for a domain more quickly and more effectively than with a general purpose language. Ideally, a well-designed DSL captures precisely the semantics of an application domain, no more and no less.

Table 1 is a partial list of domains for which DSL's have been created. As you can see, the list covers quite a lot of ground. For a list of some popular DSL's that you may have heard of, look at Table 2.[1] The first example is a set of tools known as Lex and Yacc which are used to build lexers and parsers, respectively. Thus, ironically, they are good tools for *building* DSL's (more on this later). Note that there are several document preparation languages listed; for example, LaTeX was used to create the original draft of this article. Also on the list are examples of "scripting languages," such as PERL, Tcl, and Tk, whose general domain is that of scripting text and file manipulation, GUI widgets, and other software components. When used for scripting, Visual Basic can also be viewed as a DSL, even though it is usually thought of as general-purpose. I have included one other general-purpose language, Prolog, because it

[*]Appeared as Chapter 3 in *Handbook of Programming Languages, Vol. III: Little Languages and Tools*, Peter H. Salas, ed., MacMillan, Indianapolis, pp. 39–60, 1998.
[1]Both of these tables are incomplete; feel free to add your favorite examples to them.

Scheduling	Modeling
Simulation	Graphical user interfaces
Lexing and parsing	Symbolic computing
Attribute grammars	CAD/CAM
Robotics	Hardware description
Silicon layout	Text/pattern-matching
Graphics and animation	Computer music
Databases	Distributed/Parallel comp.
Logic	Security

Table 1: Application Domains

is excellent for applications specified using predicate calculus.[2] The tongue-in-cheek comment in Table 2 regarding the application domain for the Excel Macro language points out just how powerful—and general purpose—a DSL can be, despite original intentions.

For another perspective of DSLs, it is often said that *abstraction* is the most important factor in writing good software, a point which I firmly believe in. Software designers are trained to use a variety of abstraction mechanisms: abstract data-types, (higher-order) functions and procedures, modules, classes, objects, monads, continuations, etc. An important point about these mechanisms is that they are fairly *general*—for example, most algorithmic strategies and computational structures can be implemented using either functional or object-oriented abstraction techniques. Although generality is good, we might ask what the ideal abstraction for a particular application is. In my opinion it is a DSL: the "ultimate abstraction" of a problem domain.

Advantages Programs written in a DSL have the following advantages over those written in more conventional languages:

- they are more concise
- they can be written more quickly
- they are easier to maintain
- they are easier to reason about

These advantages are the same as those claimed for progams written in conventional high-level languages, so perhaps DSL's are just *very* high-level languages?

[2]Other examples of this sort include the general-purpose functional languages Haskell and ML, which are excellent for functional specifications; we will use Haskell later as a vehicle for designing new DSL's.

DSL	**Application**
Lex and Yacc	program lexing and parsing
PERL	text/file manipulation/scripting
VHDL	hardware description
TEX, LATEX, troff	document layout
HTML, SGML	document markup
SQL, LDL, QUEL	databases
pic, postscript	2D graphics
Open GL	high-level 3D graphics
Tcl, Tk	GUI scripting
Mathematica, Maple	symbolic computation
AutoLisp/AutoCAD	computer aided design
Csh	OS scripting (Unix)
IDL	component technology (COM/CORBA)
Emacs Lisp	text editing
Prolog	logic
Visual Basic	scripting and more
Excel Macro Language	spreadsheets and many things never intended

Table 2: Popular DSL's

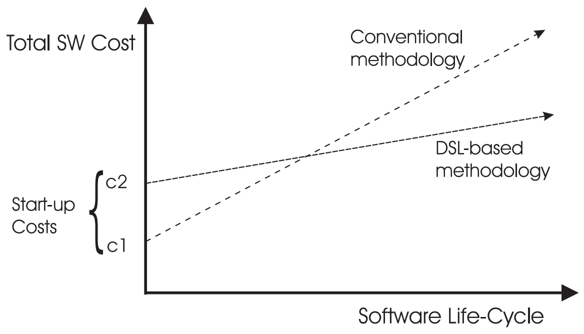

Figure 1: The Payoff of DSL Technology

In some sense this is true. But programs written in a DSL also have one other important characteristic:

- they can often be written by non-programmers

More precisely, they can be written by non-programmers who are nevertheless experts in the domain for which the DSL was designed. This helps bridge the gap (often a chasm) between developer and user, a potentially major hidden cost in software development. It also raises an important point about DSL design: a user immersed in a domain *already knows* the domain semantics. All the DSL designer needs to do is provide a notation to express that semantics.

The Payoff DSL's certainly seem to have the potential to improve our productivity as programmers, but is there some way to quantify this argument? I believe there is, as illustrated in Figure 1. Here we see that 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. And of course, if you are using an existing DSL, the initial tooling-up cost will be very low. In any case, if you are contemplating designing your own DSL, this is the chart to show to your boss.

A Few Examples Before going further, let's look at a few instances of DSL's to get a feel for their power and simplicity. For starters, Figure 2 shows a simplified version of the HTML code used to generate my home page on the WWW. Although some might not think of HTML as a programming language, it is nevertheless a notation for specifying a certain kind of computation: namely, document layout parameters.

In HTML, regions of text that are to be treated in a special way begin with a *tag* enclosed in angle brackets, and end with the same tag, but preceeded with a slash character. For example, in Figure 2, the text "`Professor Paul Hudak's Home Page`" is enclosed in a pair of "`title`" tags, which gives a title to the document, which is then used by browsers to, for example, label bookmark entries. The next line marks the beginning of the body of the page, and specifies what background color to use. Then the first line of actual text appears, rendered in the style "`h1`," which is a document *heading*. Following this text are two images (`img`), one a picture of me, one a group logo, both referenced as `.gif` files. The single commands `<p>` and `<hr>` generate new paragraphs and draw horizontal lines, respectively. The section that begins with `<ul>` is an unnumbered list, with each entry beginning with the tag `<li>`. The first entry in this list references another WWW page using what is called an *anchor*, whose tag includes the URL. The effect of this tag is that the text "A Gentle Introduction to Haskell" will be highlighted, and will take the user to the page referenced by the URL if it is clicked. The other list items are omitted, and the page closes with my address anchored in a different way: if you click on my email address, your mailer will open an out-going message with my address already loaded in the `To:` field.

There is quite a lot going on here! But I think that with my explanation you should now understand it, and that in a day's time I could teach you enough HTML that you could handle 90% of all WWW applications.

As another example, suppose we are implementing a programming language whose BNF syntax includes the following specification [ASU87]:

$$\begin{array}{lcl} \mathit{expr} & = & \mathit{expr} + \mathit{term} \;\mid \mathit{term} \\ \mathit{term} & = & \mathit{term} * \mathit{factor} \;\mid \mathit{factor} \\ \mathit{factor} & = & (\mathit{expr}\,) \;\mid \mathbf{digit} \end{array}$$

which is the standard way to define simple arithmetic expressions. Using Yacc [Joh75], this specification can be used almost directly:

```
expr    : expr '+' term
        | term           ;
term    : term '*' factor
        | factor         ;
factor  : '(' expr ')'
        | DIGIT          ;
```

Yacc will take this code and generate a parser for the little language that it represents; i.e. Yacc is a program that generates other programs. It also has several other features which allow one to specify actions to be taken for each line in the code. I also believe that, if you are familiar with BNF syntax and basic C programming, I could teach you how to use Yacc expertly within one day's time.

```
<title> Professor Paul Hudak's Home Page </title>
<body background="backgrounds/gray_weave.gif" >

<h1> Professor Paul Hudak </h1>
<hr>
<img src="hudak.gif">
<img align=top src="yale-haskell-logo.gif">
<p>
<b> Computer Science Department, Yale University </b>
<p> <hr> <p>
<h2> Functional Programming </h2>
Here I briefly describe some of my work in functional programming:
<ul>
<li> The best introduction to Haskell is the tutorial
<a href="http://www.haskell.org/tutorial"> A Gentle Introduction to
Haskell </a>.
<li> ...
<li> ...
</ul>
<address>
Paul Hudak,
<a href="mailto:paul.hudak@yale.edu"> paul.hudak@yale.edu </a>
</address>
</body>
```

Figure 2: Example of HTML Code

As a final example, let's look at some SQL code. SQL is a language for creating and querying *relational databases.* We will ignore how SQL is used to set up a database, and concentrate on what queries look like. Here is the first example:

```
SELECT  firstName, lastName, address
FROM    employee
WHERE   firstName = 'Cathy' AND birthDate = '07/04/1950'
```

This query is almost self-explanatory: it is SELECTing the first name, last name, and address FROM the employee relation WHERE the first name of the employee is Cathy AND she was born on July 4, 1950. SQL queries can be quite complex; for example they can be nested, as in the following:

```
SELECT firstName, lastName, address
FROM   employee
WHERE  salary > ALL (SELECT salary
                     FROM employee
                     WHERE firstName = 'Paul')
```

This query selects all employees whose salary is greater than that of any employee whose first name is Paul.

SQL is a powerful language for programming databases (there are other languages that are claimed to be more powerful, such as QUEL). If you are familiar with databases, my feeling is that you could learn quite a bit about SQL within a day's time, although it might take years to become an expert at using this particular DSL in the context of real-world database systems.

In the remainder of this article I will discuss the motivation of DSL's, their basic characteristics, how to design and implement them, how to embed them in existing languages, and avenues for further development. I am particulary keen on conveying the idea that you can "roll your own" DSL: you don't have to rely on existing DSL's to take advantage of this useful technology.[3]

2 The DSL Software Development Method

The basic "DSL Software Development Method" can be summarized as follows:

[3]In this regard my message is the same as that given by Jon Bentley in his excellent article on "Little Languages" [Ben86], although DSL's are not necessarily little.

1. Define your domain.
2. Design a DSL that accurately captures the domain semantics.
3. Build sofware tools to support the DSL.
4. Develop applications (domain instances) using the new DSL infrastructure.

Of course, this is not necessarily a linear process: revision, refinement, enhancement, etc. are often necessary.

The first two steps are difficult, but are the key to successful application of the methodology. If the domain itself is properly identified, the DSL design should go smoothly, especially if you have experience in basic programming language principles to begin with.

There are several ways in which one could implement a DSL. If treated as a conventional language, conventional techniques could be used: build a conventional lexer and parser based on the BNF syntax; perform various high-level analyses, transformations, and optimizations on the abstract syntax generated by the parser; and then generate executable code for some host machine. In the case of a DSL, this standard approach may be modifed in a number of ways:

1. Use Lex and Yacc (themselves DSL's), or similar tools, to facilitate the construction of the lexer and parser.
2. Use a structured editor or other programming environment generator (such as the Synthesizer Generator [Rep84]) to create the infrastructure for a more sophisticated programming environment.
3. Generate code for an abstract machine (e.g. a byte-code interpreter) rather a real machine, or generate code in another language, such as C.
4. Write an interpreter rather than a compiler.

Another way to implement DSL's is discussed in the next section.

3 Domain Specific *Embedded* Languages

Despite all of the promise, there are potential problems with the DSL methodology. To start, it may be that performance is poor: very high-level languages are notoriously less efficient than lower-level languages. If performance using conventional languages is already a problem, designing a DSL may not be the best approach. On the other hand, there are certain domains where high-level optimizations are possible on DSL programs, whose results are sufficienty complex that programming them directly in a conventional language is difficult, tedious, and error-prone. Query optimizations in the domain of databases is an example of this, and in such cases a DSL may be justified as a way to *improve* performance. In any case, there are many application domains where performance is not the bottleneck, so this argument is not a show-stopper.

Another concern is the generation of a "Tower of Babel" through the creation of a new language for every domain. This is certainly a valid concern. On the other hand, if the languages are simple enough, the problem might not be nearly as bad as one might think, and in a later section we will discuss ways to make new DSL's similar enough in look-and-feel to reduce the overhead of learning many new languages.

A final concern is the potential for unacceptable start-up costs: design time, implementation, documentation, etc. It can be fairly difficult to design and implement a programming language from scratch: a 2-5 year effort is not uncommon. Moreover, there's a good chance that we won't get it right the first time. The DSL will evolve, and we will experience all of the difficulties associated with that evolution. To state this concern in concrete terms, 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?

There are dangers lurking in every software design methodology, and there are no silver bullets, of course. We must understand the benefits as well as the limitations of whatever methodology we are using, and proceed with caution.

In the case of the DSL methodology, I would like to use the rest of this article to discuss several techniques that can greatly alleviate most of the problems addressed above. These techniques rest on two key thoughts:

- 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).
- Building on this base, we can then concentrate on semantical 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.

In the following sections I will elaborate on these ideas.

3.1 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 and parsers from scratch. On the other hand, they are still rather tedious to use, and in any case 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 doesn't matter—I believe that it does—but rather places syntax in proper perspective.

However, when one focuses on semantical issues, many of the details still don't matter much. Even a deep issue such as the evaluation order of arguments is often something that people can adjust to, as long as they know exactly what it is for the language they are using. The precise behavior of variable-binding constructs, pattern-matching rules, endless details in type systems, major differences in module functionality, etc. are examples of debates that rage in the programming language design community. Other examples of semantical minutiae include names of pre-defined functions, the lexical order of their arguments, exactly how they behave under all circumstances, and an endless list of similar issues concerning the functionality of the software libraries that are essential to making a programming language practical.

The bottom line is that, once a programming language is chosen, people get the job done, and they usually well appreciate the high-level language that they are using.

3.2 DSELs Inherit Language Features

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. Aside from the obvious advantage of being able to reuse many ideas and artifacts, DSELs have certain advantage 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 not usually 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.

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. Indeed, for 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.

In summary, the DSEL approach creates a rich infrastructure that:

1. Allows for *rapid* DSL design; if nothing else, it can be viewed as a way to *prototype* a DSL.
2. Facilitates *change*, whether for experimentation, fault correction, or design evolution.
3. Provides a familiar *look and feel*, especially for several different DSL's embedded in the same language. In other words, it reduces the size of the Tower of Babel.
4. Facilitates *reuse* of syntax, semantics, implementation code, software tools, documentation, and other related artifacts.

Of course, there is danger lurking in this approach as well: we may find that our DSL becomes limited by the power and prejudices of the underlying "host" language. It is important to design the DSL abstractly first, and then search for a suitable host.

Implementing a DSEL can be achieved by writing a pre-processor for the host language (and possibly a post-processor of program output), or by directly modifying the host language implementation. The former is more desirable since it removes dependency on other evolving systems. In many cases the embedding can be achieved without any pre-processor at all: for this to work, the host language needs to be suitably rich in syntax and semantics (we will see examples of this shortly).

In the remainder of this article I will describe the results of using the functional language *Haskell* [HPJWe92] to build DSELs. Haskell has several features that make it particularly suitable for this—in particular, higher-order functions, lazy evaluation, and type classes—but other languages could conceivably be used instead. On the other hand, there are features that don't exist in any language (to my knowledge) that would make things even easier; there is much more work to be done.

3.3 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 many domains, including lexer and parser generation, graphics, animation, simulation, concurrency, computer music, GUI construction, scripting, hardware description, VLSI layout, pretty printing, and geometric region analysis. The latter domain—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]) 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 quite easy to incorporate.

The resulting notation is not only easy to design, it's also easy to use and reason about. Figure 3 shows some of the code to give the reader a feel for its simplicity and clarity. 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.

(A few notes on Haskell syntax: a type such as `Point -> Bool` is the type of functions that map values of type `Point` to values of type `Bool`. In Figure 3, the name `Region` is given to this type. A statement such as `circle :: Radius -> Region` declares that the value `circle` is a function from type `Radius` to type `Region`. Function application in Haskell is written `f x y`; this is the same as `f(x,y)` in many other languages. Also in Haskell, any function can be used in infix position by enclosing it in back-quotes. Thus ``p `inRegion` r`` is the same as `inRegion p r`.)

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 creat this simple DSL. For example, the definition of `(/\)` is given by:

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

which can be read: "the intersection of `r1` and `r2` is a region which, when applied to a point `p`, returns the conjunction of `r1` applied to `p` and `r2` applied to `p`." Or more abstractly: "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 clean underlying semantics. The key point is that one can reason directly *within the domain semantics*, rather than within the semantics of the programming language. In the NSWC experiment, several properties of the DSEL were straightforwardly proved that would have been much more difficult to prove in most of the competing designs. For example, to prove associativity of region intersection:

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

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

-- Given suitable definitions of "circle", "outside", and /\:
circle   :: Radius -> Region
            -- creates a circular region with the 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 a function to generate an annulus:
annulus      :: Radius -> Radius -> Region
annulus r1 r2 = outside (circle r1) /\ circle r2
```

Figure 3: Example of a DSEL for a Naval Application

```
    (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          (unfolding of /\)
  = (r1 p && r2 p) && r3 p        (unfolding of /\)
  = r1 p && (r2 p && r3 p)        (associativity of &&)
  = r1 p && (r2 /\ r3) p          (folding of /\)
  = (r1 /\ (r2 /\ r3)) p          (folding of /\)
```

The *unfolding* of a function definition means replacing an instance of the left-hand side with the right-hand side, whereas the *folding* of a function definition means replacing an instance of the right with the left.

This simple use of formal methods results in a rich *algebra* that captures the domain semantics quite nicely. This will be elaborated on in the next section.

```
-- 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 4: A Simple Graphics DSEL

4 Modular Semantics

In a later section I will 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], a DSEL for building "functional reactive animations." We begin with some simple operators for manipulating graphical objects, or "pictures," as shown in Figure 4.[4]

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 then be proven.

4.2 Simple Animations

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

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

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

But in fact *many* sorts of things could be time varying. Thus we will 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:

```
(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, but we note that the vector and color arguments might also 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)
```

Finally, we define a new function to return the current time:

```
time t = t
```

With this simple transformation we can now express continuous-time animations. For example, let's first define a couple of simple utility behaviors. The first varies smoothly and cyclically between -1 and +1:

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

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
```

Now let's create a very simple animation: a red, pulsating ball.

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

We can also develop a rich algebra of animations. 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 animation `anim` with a copy of itself just above itself 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)` in Haskell denotes the composition of the functions `f` and `g`.

4.3 Reactive Animations

For the third and last layer of our semantic structure, we will 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 then has the form:

```
    b1 ‘until‘ e => b2
```

which can be read: “behave as `b1` until event `e` occurs, then behave as `b2`.” For example, here is a circle that changes color everytime the left mouse-button is pressed:

```
  colorB (cycle red green blue) circle
  where cycle c1 c2 c3 =
              c1 ‘until‘ leftButtonPress => cycle c2 c3 c1
```

Again we find that the previous algebraic semantics 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.

5 Advanced Support For DSEL’s

In this section I will describe some recent research and development efforts that promise to make the DSL/DSEL methodology even more attractive for industrial-strength software development.

5.1 Modular 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 syntax. In some sense it is just a notation; its semantics is captured by an *interpreter*. This permits another opportunity for modular design, in turn facilitating 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 several researchers [Mog89, Ste94, Esp95, LHJ95, LH96, Lia96] has led to some key principles on which one can base such modular interpreters and compilers. The use of *monads* [PJW93, Wad90] to structure the design was critical.

This 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, one can build a series of languages and interpreters that begin with a small calculator language (just numbers), then a simple first-order language

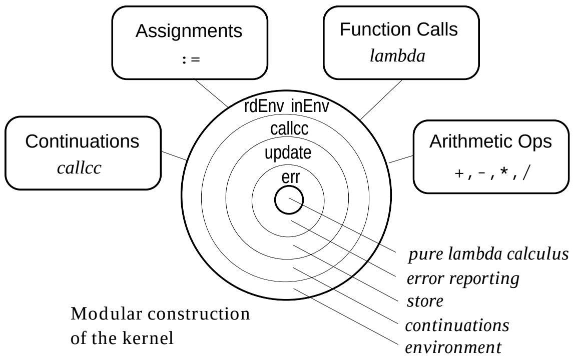

Figure 5: Modular monadic interpreter structure

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 5. 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. This idea has been used to implement traditional compiler optimizations [LH96, Lia96], but the same techniques could be used for domain-specific optimizations.

A conventional interpreter maps, say, a term, environment, and store, to an answer. In contrast, a modular 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 interpreter 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 undefined value). 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, tracing, profiling, and even references and assignment, to our interpreter. 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.

5.2 Language Tools and Instrumentation

Despite the importance in software development of language tools such as debuggers, profilers, tracers, and performance monitors, traditionally they have been treated in rather *ad hoc* ways. I believe that a more disciplined approach to designing such tools will benefit the software development process. Indeed, it is possible to develop a methodology for tool generation that shares much with previously identified goals: it is highly-modular, domain-specific, and evolvable. Under this scheme, tools can be layered onto the system without affecting each other; changes and additions are thus easily accomplished. A tool specified in

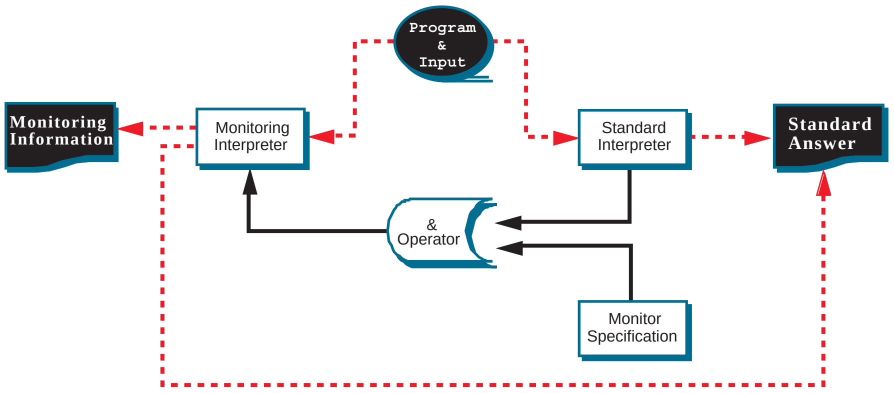

Figure 6: System diagram

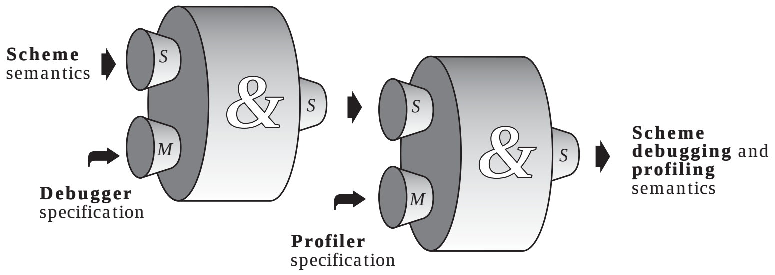

Figure 7: Composing monitors

this framework can be automatically combined with the corresponding standard semantics to yield a composite semantics that incorporates the behaviors of both. Figure 6 is a flow diagram for the overall methodology, and Figure 7 shows its compositional nature.

5.3 Partial Evaluation.

In order to use DSELs and their corresponding modular interpreters in a practical sense, we can use program transformation and partial evaluation technology to improve performance. For example, we can use partial evaluation to optimize the composed interpreters described previously in two ways: (1) specializing the *tool generator* with respect to a *tool specification* automatically yields a *concrete tool*; i.e. an interpreter instrumented with tool actions, and (2) specializing the *tool itself* (from the previous step) with respect to a *source program* produces an *instrumented program*; i.e. a program with embedded code to perform the tool actions. Figure 8 provides a useful viewpoint of these two levels of optimization.

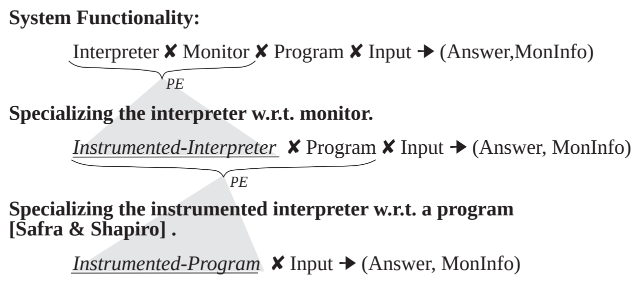
System Functionality:
Interpreter ✘ Monitor ✘ Program ✘ Input ➔ (Answer,MonInfo)
PE
Specializing the interpreter w.r.t. monitor.
Instrumented-Interpreter ✘ Program ✘ Input ➔ (Answer, MonInfo)
PE
Specializing the instrumented interpreter w.r.t. a program [Safra & Shapiro] .
Instrumented-Program ✘ Input ➔ (Answer, MonInfo)

Figure 8: Partial evaluation optimization levels

The current state-of-the-art in partial evaluation technology is unfortunately not robust enough to perform these transformations directly on Haskell programs. The dramatic improvement in performance that can be achieved, however, is providing the impetus to create partial evaluation tools that will satisfy this need.

6 Conclusion

I have described a methodology for designing and implementing domain-specific languages. Some of the techniques to do this are well-known, being similar to techniques for implementing conventional programming languages. Others are much newer, yet many of these have been used well enough to give us confidence of their success. The notion of an *embedded* DSL is especially attractive, due to its simplicity and power.

I urge the reader to search for opportunities to use DSL's and to create new ones—however small—when the need arises. A well-designed DSL should capture the essence of one's application domain, and in that sense there is no better way to structure one's software system. There are no hard-fast rules for designing a DSL, but the following guidelines may be useful:

1. Use the KISS (keep it simple, stupid) principle.
2. "Little languages" are a Good Thing; read Jon Bentley's article.
3. Concentrate on the domain semantics; do not get hung up on syntax.
4. Don't let performance dominate the design; and don't let the design dominate performance.
5. Prototype your design; refine and iterate.
6. Keep the *end user* in mind; remember that Success = A Happy Customer.

References

[ASU87] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1987.

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

[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.

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

[Esp95] David Espinosa. *Semantic Lego.* PhD thesis, Columbia University, 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 Non-strict 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.

[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.

[Lia96] Sheng Liang. *Modular Monadic Semantics and Compilation.* PhD thesis, Yale University, Department of Computer Science, November 1996.

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

[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.