Building Domain-Specific Embedded Languages

Paul Hudak
Department of Computer Science
Yale University

June 3, 1996

I have believed for a very long time that *abstraction* is the most important factor in writing good software. As programming language researchers we design, and as software engineers we are trained to use, a variety of abstraction mechanisms: abstract data types, higher-order functions, monads, continuations, modules, classes, objects, etc. Particular languages support some of these mechanisms well, others not so well. 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 *programming language* that is designed precisely for that application: one in which a person can quickly and effectively develop a complete software system. It is not general at all; it should capture precisely the semantics of the application domain—no more and no less. In my opinion, a domain-specific language is the "ultimate abstraction."

But we know all too well how difficult designing and implementing languages is, and we can be pretty sure 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. So in fact the notion of a domain specific language might not be very practical. Or is it?

In this position paper I will outline several techniques that I believe can lead to the effective use of this methodology. It begins with the assumption that we really don't want to build a 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: viz. the *interpreter* of the language. Interestingly, we'll see that abstraction now kicks in at this leta-level: we can use abstraction techniques to build interpreters that are themselves easy to understand, highly modular, and straightforward to evolve.

In the remainder of this paper I will describe the results of using the functional language Haskell to build DSELs. Haskell has several features that make it particularly suitable for this, but other languages could also be used. 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.

Domain Specific Semantics It is surprisingly straightforward to design a DSEL for many specific applications. We have done so already using Haskell in several domains: parser generation, graphics, animation, simulation, music composition, and geometric region analysis, to

```
-- 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 region with given radius
outside  :: Region -> Region            -- the logical negation of a region
(/\)     :: Region -> Region -> Region  -- the intersection 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 1: Example of a DSEL for a Naval Application

name a few. The latter domain—geometric region analysis—came about through an experiment conducted jointly by Arpa, 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 quite easy to incorporate.

The resulting notation is not only easy to design, it's also easy to use and reason about. 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. In Figure 1 we highlight some of the code to give the reader a feel for its simplicity and clarity.

Finally, the DSEL approach is highly amenable to formal methods, for many of the reasons already mentioned. 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 we straightforwardly proved several properties of our DSEL that would have been much more difficult to prove in most of the competing designs.

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 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 Moggi, Espanol, and Steele [Mog89, Ste94, Esp93] laid the groundwork for our recent effort at producing a truly

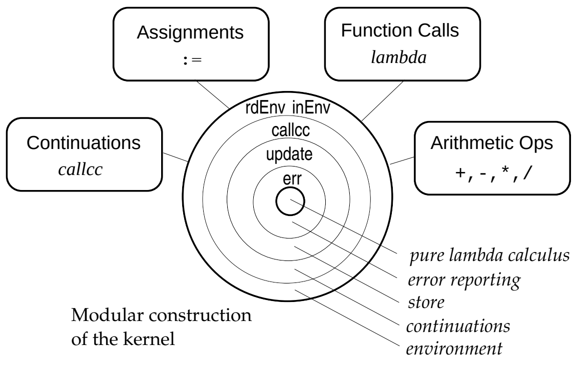

Figure 2: Modular monadic interpreter structure

modular interpreter for a non-trivial language [LHJ95], and basing modular compiler construction technology on it [LH96, Lia96]. 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 2. 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, 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 monadic interpreter such as ours 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 our 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:

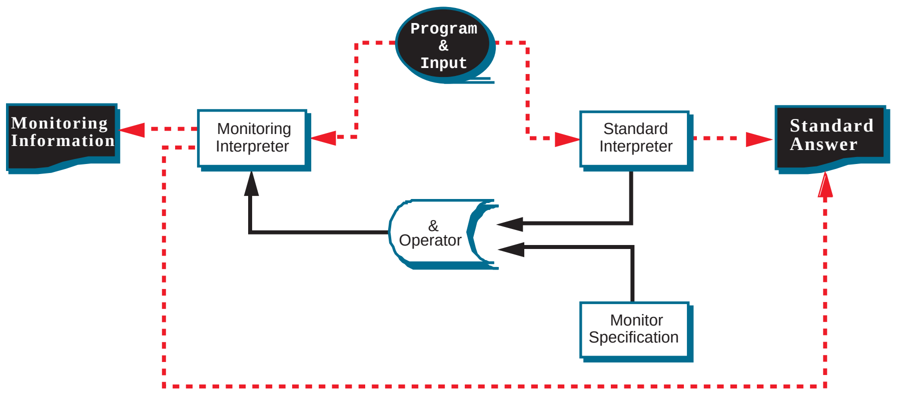

Figure 3: System diagram

```
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.

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 we have a methodology for tool generation that shares much with previous 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 our framework can be automatically combined with the corresponding standard semantics to yield a composite semantics that incorporates the behaviors of both. Figure 3 is a flow diagram for the overall methodology, and Figure 4 shows its compositional nature.

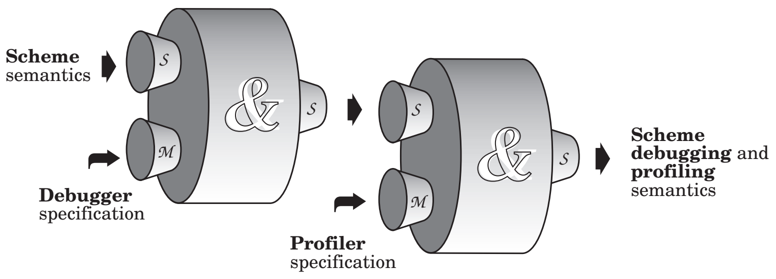
Scheme semantics
Debugger specification
Profiler specification
Scheme debugging and **profiling** semantics
&
S
M

Figure 4: Composing monitors

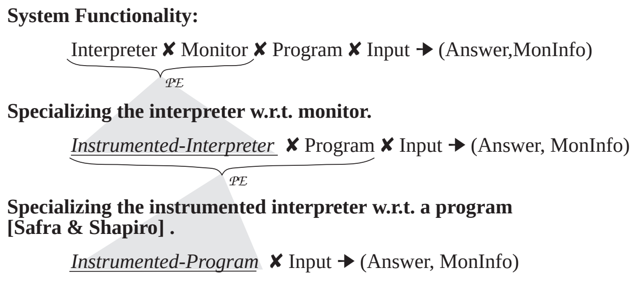
System Functionality:
Interpreter ✘ Monitor ✘ Program ✘ Input → (Answer,MonInfo)
PE
Specializing the interpreter w.r.t. monitor.
Instrumented-Interpreter ✘ Program ✘ Input → (Answer, MonInfo)
PE
Specializing the instrumented interpreter w.r.t. a program [Safra & Shapiro] .
Instrumented-Program ✘ Input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

Figure 5: Partial evaluation optimization levels

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 5 provides a useful viewpoint of these two levels of optimization.

We have used existing partial evaluation techniques to do this, but it was painful. I feel that more user-friendly techniques are needed. In particular, in contrast to a fully-automated approach, a *semi-automated* approach would have two advantages: First, automatic approaches have not matured enough in recent years to give us the confidence we need to meet our goals. Second, I think that it's important for the user to have better, more explicit control over the transformation process. Reasoning about the behavior of fully-automated systems can be difficult, and it gets worse as the sophistication of the automation increases.

References

[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. Last modified October 27, 1993; further changes specified by J. Caruso are described in "Addendum to Prototyping Demonstration Problem for the Prototech HiPer-D Joint Prototyping Demonstration Project," November 9, 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.

[Esp93] David Espinosa. Modular Denotational Semantics. Unpublished manuscript, December 1993.

[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.

[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*. 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).

[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.