The Art of Interpretation for Domain-Specific Embedded Languages

Saswat Anand[1], Siau-Cheng Khoo[2], Dana N. Xu[2], and Ping Zhu[2]

[1] College of Computing, Georgia Institute of Technology
[2] School of Computing, National University of Singapore

Abstract. Domain-specific embedded language (DSEL) has become a convincing approach in managing the complexity of domain-specific applications. Its dependence on a host language enables domain experts to build programs at faster speed. Building DSEL on a host language with interpretive ability also provides a familiar calculator approach to domain experts, many of whom are programming novice. However, interpretive overhead can be a concern for the use of DSEL. In contrast with other research that investigates the compilation process of DSEL, we propose in this paper a framework for designing Haskell-based DSEL to attain efficiency in the interpretive environment. We lift the design of interpreter beyond the level of syntax-based interpretation, thus overcoming the notorious syntax-analysis problem with DSEL. One main technical contribution is the systematic and creative use of Haskell's overloading mechanism to achieve domain-specific program analysis at interpretative time. In addition, we advocate the use of abstract data types, not just monadic computing, to achieve various domain-specific optimizations. We illustrate these features by designing a DSEL that supports technical analysis of financial forecasting.

1 Introduction

Domain-specific embedded language (DSEL) has become a convincing approach in managing complexity of domain-specific applications. Its dependence on a wide-spectrum host language enables DSEL designers to rapidly construct prototypes, thus shortening interactive software development cycle.

From the viewpoint of many users and domain experts – a majority of whom are programming novice, a DSEL constructed on top of a host language with interpretive ability, such as Haskell, SML, *etc.*, has even greater appeal. Such interpretive DSEL simulates the use of calculator, which immensely increases the friendliness of the language.

In some domains, writing interpretive programs is almost inevitable, as use of calculator has become part of the domain culture. As an example, in financial sector, simple technical indicators used for stock forecasting are commonly defined and manipulated using formulae written in speadsheets, such as Excel, *etc.* Here, the interpretive nature of technical indicator can be easily preserved by an interpretive DSEL. It will be repulsive for domain experts to consider compilation of each technical indicator definition before its use.

Despite its attractiveness, use of interpretive DSEL has been plagued by performance efficiency problem, just like any interpretive language. For years, researchers have focused on compilation approach to address this problem, at the expense of upholding the user-friendliness of interpretive DSEL [8, 5, 12]. This indirectly highlights the shortcoming of the state of the art of DSEL, as it requires users of the application domains to shift their mindset about DSEL operation mode.

In this paper, we propose to invest software design effort in the construction of an interpretive system which boosts the performance of the system without resorting to compilation.

Our design emphasizes on both flexibility and efficiency. We follow the traditional approach of software design by employing *abstract data types* to ensure high cohesion yet low coupling of various components of our system. In addition, we introduce a *one-time analysis* of DSEL programs, performed at interpretive time, to uncover optimization opportunity. To our knowledge, there have not been any DSEL interpreters which are built from these two thrusts. One plausible reason is the notorious problem with performing syntax analysis on DSEL. Specifically, DSEL language carries the syntax baggage of its underlying host language; this makes performing syntax analysis on DSEL a delicate task, as the DSEL designers may need to perform similar analysis over the entire host language.

Another plausible reason for not investing on abstract data type is the perception that interpreter of a language must be syntax-directed. Coupled with the belief that interpreters are mainly used during rapid prototyping, little effort has been spent on investigating optimization opportunities.

To demonstrate the feasibility of our ideas, we choose a Chart Pattern Language (CPL) as our target DSEL building on top of Haskell [2, 3]. Part of the CPL, the technical indicators, is akin to the definition of formulae in spreadsheet. For instance, an three day close-price moving average can be defined as

```
close + close #1 + close #2
```

where `close` refers to the closing price of today, and `close#1` and `close#2` refer respectively to the closing prices of yesterday, and of the day before. It is common for a financial trader to call up a calculation of this moving average price for a company, say IBM, by simply entering the command:

```
eval (close + close #1 + close #2) ‘‘IBM’’
```

Such an interactive environment is important to the usability of the system, and users will be put off using it if they have to compile the above technical indicators, and many others, before getting them evaluated against various companies.

In this paper, we demonstrate a framework for constructing an efficient version of such an interpreter.

The inclusion of performance consideration inevitably requires us to consider alternate evaluation order of DSEL programs. Hence, we must ensure certain

degree of flexibility during interpretation. We accomplish this by encapsulating *monadic computation* in abstract data types (ADT). As commonly known, ADT enables different, more efficient data structures to be employed for performance consideration, while providing a friendly front end to the user. This is particularly crucial for a DSEL/DSL language, whereby the users are more concern with the usability of the language than its efficiency coding. The role of ADT is primarily to provide such a *separation of concern*.

While such an introduction of abstraction adds interpretation overhead to the final product, we will show that its benefits, in terms of both the ease of use and the realization of performance optimization, easily offset the cost.

Some of the performance boosters are dictated by analyses of DSEL programs. As memtioned earlier, these analyses are seldom done in the interpretive environment, due to the incompatibility between syntax-directedness and design of DSEL.

Our main technical contribution in this paper is to develop analyses for DSEL that do **not** directly perform syntax analysis. Instead, we leverage on the *Haskell class system* to design our analysis. Consequently, an analysis is viewed as an alternate interpretation of DSEL programs. Our implementation can be viewed as a realization of *abstract interpretation* for DSEL using Haskell overloading mechanism. Indeed, we will show how a general abstract interpretation framework for DSEL programs can be implemented.

The outline of the paper is as follows: We first provide an overview of a DSEL, the Chart Pattern Language (CPL), as our running example. In Section 3, we describe the organization of Haskell classes to support analysis of CPL programs. Section 4 highlights the novel view of monads as abstract data types. Section 5 details the performance gained from this construction. This is followed by a discussion on alternate implementation of interpretive DSEL, in particular, combinator-based DSEL. We then conclude the paper by discussing the related works.

2 CPL: A Running Example of a DSEL

We use Chart Pattern Language (CPL) as a DSEL thoughout the paper to illustrate our approach. CPL is a domain specific programming language embedded in Haskell. It is used for programming analysis and forecasting techniques for stock market. There is one school of thought, which thinks that price of stocks moves in patterns and these patterns can be used to forecast future price. CPL users belong to this school, who want program these patterns for their analysis. An example of one such pattern is show in Figure **??** and its CPL definition in Figure 2. The pattern definition is read as: a head-and-shoulder instance is composed of six patterns (three `up`s and three `down`s back to back and if `a`, `b`, `c`, `d`, `e`, `f` and `g` are the starting point, joining points of these patterns and end point respectively (referred to as landmarks in CPL) then six constraints involving these landmarks must be satisfied. `close` and `high` functions return the closing and high price of a stock for a given day.

For those unfamiliar with technical analysis, we should mention that the following definition of the pattern is just one of the many that can be written for head-and-shoulder. There is no consensus on how a particular pattern is defined – although the basic definition that defines the overall shape of the pattern will be same there can be mutitudes of differences in the details. And that is precisely the reason why we need a programming language so the user has the liberty to define a pattern the way he likes. The advantages of embedding CPL in a declarative

```
headShoulder1 = let pat = up >.> down >.> up >.> down >.> up >.> down
                    cf  = \(Lms ([a,b,c,d,e,f,g])) ->
                                [close a == close c,     -- 2
                                 close c == close e,     -- 5
                                 close e == close g,     -- 7
                                 close d > high b,       -- 4
                                 close d > high f,       -- 6
                                 abs(close b - close f) < 0.2 * close f]
                                                         -- 6
                in pat ?? cf
```

Fig. 1. CPL definition of head-and-shoulder pattern

language like Haskell is clearly evident from the above definition. The syntax of CPL is very high-level and easy to use, even for a stock analyst without any programming background. In addition to aesthetics, CPL is as expressible and powerful as Haskell. Although CPL, like any other embedded language, has a nice syntax and sound semantics, the picture is not as rosy in the pragmatics section, which is a serious limitation. After the user defines a pattern, normally she would want to search instances of that pattern that satisfy this definition in a price history that is usually huge. So how efficiently CPL pattern definitions can be evaluated is of critical importance.

Figure 2 shows the optimal version of the preceeding definition of head-and-shoulder.

To understand why the latter definition is efficient, we need to know how this pattern definition is evaluated. As show in the Figure **??**, for a pattern definition of the form $p??f$ we first evaluate all instance of pattern p and then filter out those instances from the result that do not satsify all the constraints specified in f. But due to lazy evaluation finding instances of pattern p and evaluation of constraints occur in an interleaving manner and thus makes the search efficient. For example if an instance of `up >.> down` do not satisfy the constraint `close a == close c` of the head-and-shoulder definition, the instance can not be extended to form a head-and-shoulder. Lazy evaluation produces this behavior for free. As we can see here, the order in which constraints are evaluated plays a

```
headShoulder1 = let pat = up >.> down >.> up >.> down >.> up >.> down
                    cf  = \(Lms [a,b,c,d,e,f,g]) ->
                                [close a == close c,     -- 2
                                 close d > high b,       -- 4
                                 close c == close e,     -- 5
                                 close d > high f,       -- 6
                                 abs(close b - close f) < 0.2 * close f]
                                                         -- 6
                                 close e == close g,     -- 7
                in pat ?? cf
```

Fig. 2. Optimal CPL definition of head-and-shoulder pattern

critical role in how efficiently the patterns can be evaluated. By default, the constraints are evaluated in the order in which they are specified, which may not always be the best order. For example, if in the list of constraints, the first constraint involves the very first and the last landmarks of a pattern (eg. `a` and `g` for head-and-shoulder), for evaluation of this constraint we need to find a complete instance of the pattern, so that we have values for the required landmarks. (We assume that pattern is evaluated left to right, but the other way is also possible, in which case similar reasoning would still hold.) On the other hand, if the first constraint involved first and second landmarks of the pattern and that constraint was not satisfied, it would have saved us from evaluating any further for that particular instance. In other words, it is always efficient to evaluate a constraint that has a lower "depth". If a constraint involves i^{th}, j^{th}, k^{th}, ... landmarks of a pattern, depth of that constraint is $max(i, j, k \ldots)$. In Figure **??** and Figure **??** the numbers shown to the right of each constraint represents its depth. The second definition of head-and-shoulder specifies the constraints in the optimal order. In section **??** we describe an optimization technique, which aims to find the depths of constraints and evaluate the constraints in increasing order of their depths.

– should give the semantics of ?? operator from ICFP paper. –

Optimizing DSEL is a challenging problem when domain-specific entities are modelled as functions, like our constraint function that takes a list of landmarks and returns a list of booleans. Because functions are like black boxes – we have no way to analyze and modify the body of a function unless we work on it as a piece of code. In section **??** we offer a solution to this problem based on abstract interpretation and class overloading.

3 Defining Program Analysis

One of the most technically challenging tasks in dealing with an interpretive DSEL is to perform analysis of DSEL programs. The problem stems from seek-

ing a handle for manipulating the program syntax. If a DSEL is totally defined using algebraic data types, then one can perform, during interpretation, syntax analysis over these algebraic data types. This pre-processing technique is readily described in some elementary textbook on program interpretation (*eg.* Chapter 4.1.7 of [1]). In practice, however, many DSEL programs are (partly) constructed using function representation. Since these functions are usually treated as "black-boxes" (*ie.* closures) during interpretation, there is no handle for syntax manipulation.

In CPL, constraints associated with patterns are represented using function of type `Lms ([a] -> [b])`. While this representation is both elegant and natural (as the formal arguments provide a means to name the landmarks of a pattern), it inhibits direct manipulation of constraint list encapsulated in the function. As mentioned in Section 2, re-ordering of these constraints is required so that constraints can be solved in the same order as the construction of pattern instances.

We present a solution to this problem through Haskell's overloading mechanism. To illustrate our solution, let's consider the following constraint function:

```
\ [a,b,c,d] -> [ close b > close c,          -- (2)
                 high c = high d,            -- (3)
                 open a - open d < 10 ]      -- (1)
```

This constraint is applicable to any pattern comprising three primitives, such as `(up >.> down >.> up)`. For a reason to be explained in the latter section, pattern instances are constructed "backwards" such that sub-patterns at the right end are constructed before those at the left end. The first constraint above involves second and third landmarks. This means that the constraint can be evaluated when a partial instance of the above pattern – an instance of `(down >.> up)`, more precisely – has been constructed.

Using the same reasoning, we find that second and third constraints can be evaluated when the third and the first landmarks have respectively been instantiated.

Informally, in order to find out when a constraint should be evaluated during partial instantiation of patterns, we first represent the landmark variables by their positions in the list of landmarks. We call these positions *indices*. Thus, `[a,b,c,d]` is replaced by the index list `[1,2,3,4]`. Next, operations over landmarks are treated as *operations over indices*. Thus, for each constraint, we find the *minimum* of those indices involved. In the above constraint function, we replace `a ... d` by `1 ... 4` respectively, and the analysis of the three constraints yields the indices `2`, `3` and `1` respectively. Thus, during the "backwards" construction of pattern instances, it is more efficient to solve the second constraint first, followed by the first constraint, and then the third constraint.

To incorporate this index analysis into our system, we introduce the following *trick data constructor* to represent indices:

```
data Lmk = T Int deriving (Show, Eq, Typeable, Data)
```

We dictate that *all constraint operations should be overloaded* to take in both actual landmarks and indices and produce a boolean list (for constraint solving)

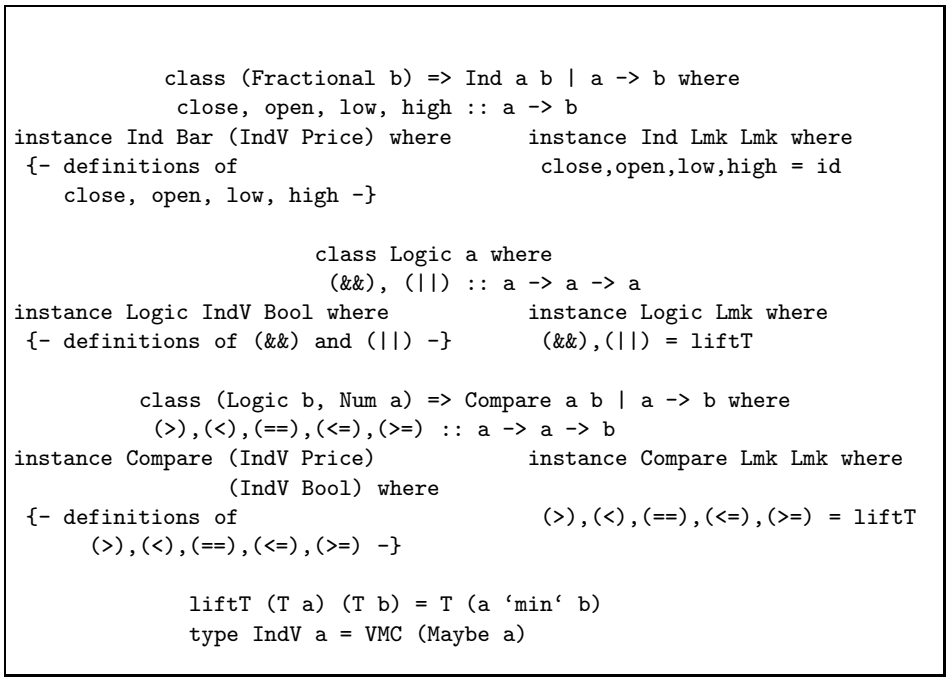

Fig. 3. Simplified Haskell Classes and Instances for Constraint Operators

and a index list (for analysis) respectively. Figure 3 briefly depicts the Haskell classes defining these overloading operators.

Describe how the analysis is called during interpretation.

3.1 Defining Abstract Functions

It is conducive to view the instances of operations defined with respect to a DSEL as a *standard interpretation*, and view those defined for an analysis as an *abstract interpretation*, *a.la* Cousot's [4]. However, there are some limitations to this approach that deserve closer investigation.

Firstly, this approach requires all Haskell primitives to belong to some Haskell classes. This is certainly not the case. Specifically, conditional constructs, such as `if`, is not defined in any Haskell class. Consequently, `if`-expression cannot be overloaded, and its evaluation order cannot be modified during abstract interpretation.

Thanks to the lazy evaluation strategy of Haskell, we can overcome this problem by defining a special conditional function at the DSEL level, as follows. Let `L` and `A` denote the datatypes used by DSEL and the analysis respectively. We have:

```
class Cond a where
```

```
                    cond :: forall c . a -> c -> c -> c
 instance Cond L where            instance Cond A where
  cond a b c = ...                 cond a b c = ...
```

Next, analysis of recursive functions has to be done by fixpoint iteration over an abstract lattice. This is handled by (1) defining a class containing lattice operators, and (2) restricting the representation of recursive function definition by *fixpoint operator*, `fix`. From point (2) above, we see that recursive funnction has to be defined in terms of *functional*, so that calls to the recursive function will be expressed by calls to `fix`. Figure 3.1 defines this solution.

```
class Eq a => Lattice a where
 elems :: [a]          -- finite list of lattice elements
 bot :: a
 lub :: a -> a -> a
 glb :: a -> a -> a
 leq :: a -> a -> Bool
 fp  :: (a -> a) -> (a -> a) -> [a] -> (a -> a) -- fixpoint computation
 fp f h data = let g = (f h) `leq_f` h
               in  if leq_f(g,h,data) then h else fp f g data
   where leq_f f g = \ x -> (f x) \lub (g x)
         leq_f(f,g,ds) = every (\ x -> (f x) `leq` (g x)) ds

class Fix a where
 fix :: (a -> a) -> a -> a
instance Fix (L -> L) where          instance Fix (A -> A) where
 fix f = f (fix f)                    fix f = fp f bot elems
```

Fig. 4. Lattice and Fixpoint operations

In summary, for general abstract interpretation (AI) to work properly in the overloading approach, every construct that is manipulable by the AI must have a handle (such as `cond`, `fix`, *etc.*) at the user level, so that different instances of handles can be defined. The onus is on DSEL designers to ensure that handles are accessible by the user.

where does SPJ's boilerplate scrapper technique comes in?

4 Monads as Abstract Data Types

Use of Abstract data types in software development is prevalent. Its role is also well-defined: To separate the implementation concern from the use of a piece of data. Unfortunately, this concern is usually raised in the design of interpreters,

probably due to the perception that interpreters primarily meant for rapid prototyping, and therefore designers are less concern about its implementation detail.

In the event that interpreter is the main interactive tool for language users, it is important that its design and development be adhered to good software engineering practice.

Performance efficiency is usually associated with in-place update, and in the context of Haskell, this is realized via *monadic computing* [7, 11]. However, monads are typically viewed as an abstract model of programming language [9, 10]. This is different from treating monads as abstract data type. Specifically, monads have mainly been used to hide some data, such as state information, from the user. They have not been used to provide a data structure at implementation level that is *distinct from the data structure perceived by the user*. To our knowledge, no work has been done in relating monads to abstract data types.

To demonstrate the need for an abstract data type, we turn to the definition of patterns in CPL. Ignoring constraint-function component, we provide a user-level algebraic data type for pattern construction, as follows:

```
data PatD = Up | Down | Fby PatD PatD
```

This data structure enables users to form pattern easily. Treating `Fby` as `>.>`, one can construct a simple head-and-shoulder pattern as follows:

```
Up >.> Down >.> Up >.> Down >.> Up >.> Down
```

From implementation viewpoint, however, searching instances of an `up` (resp. `down`) pattern that is sandwiched between two `down` (resp. `up`) pattern can be much simpler than that of a non-sandwiched pattern. This is because a sandwiched pattern assures that the end points of any of its instance must be extrema points. Consequently, there can only be at most one instance per extrema point for such a sandwiched pattern.

The above observation suggests a different algebraic data type for patterns at implementation level, to highlight different version of primitive patterns. Thus, at the implementation level, the corresponding data structure is

```
data PatI = UpI | DownI | FbyI PatI PatI | EUpI | EDownI
```

and the corresponding head-and-shoulder pattern, with slight abuse of the operator `(>.>)`, is represented at implementation by:

```
UpI >.> EDownI >.> EUpI >.> EDownI >.> EUpI >.> DownI
```

5 Performance

In this section, we report the performance of different optimizations on pattern searching as well as memoization for computation of technical indicators. All performance statistics presented in this paper are run under windowXP in a desktop PC with Pentium(R)4 CPU 2.0GHz and 512MB of RAM. For pattern searching, we use 10 years of daily stock price histories of 40 companies. For computation of a technical indicator, we use 10 years of daily price of 1 company.

Table 1. Timing for pattern searching (in sec)

Pattern	Unoptimized	datatype	backtracking
head-and-shoulder	112.84	45.48	36.76
hill	75.44	42.22	39.63
up3	hangs	78.04	65.10

Table 2. Timing for computation of technical indicator (in sec)

Technical Indicator	without memoization	with memoization
mvg (mvg close 10) 20	7.63	2.06
mvg (mvg close 20) 50	39.13	3.44
mvg (mvg close 50) 100	hangs	6.72

6 Combinator-based CPL

7 Related Work

The idea of virtual machine in our approach is originated from John Launchbury's work on constructing a monadic interpreter for a simplified DSL: a graph-travesal language describing depth-first graph traversals. He defined some monadic virtual machine operations to access, update graph's state and traverse graph. Based on these primitive functions, he described the interpreter in a much more compact manner. We extend Launchbury's work by adding the optimization function in our virtual machine.

7.1 Type-based overloading

The idea of type-based overloading used in our abstract interpretation is borrowed from Simon Peyton Jones's "Scrap your boilerplate" approach to generic programming. He observed that in programs traversing data structures built from rich mutually-recursive data types, there will be a great deal of "boilerplate" code that simply walks the structure, hiding a small amount of "real" code that constitutes the reason for the traversal. He defined a generic mapping operation $gmapQ : (gmapQ\ f\ t)$ applies the polymorphic function f to each of the immediate children of the data structure t (these data structures may have different types so f must be a polymorphic function) and returns a list of these results. This technique thus leaves the programmer free to concentrate on the important part of the algorithm: defining the "real" type-specific code f. Simon Peyton Jones demonstrate a concrete example of this type-based overloading function f: f will perform certain desired analysis or transformation on some nodes with some specific types while give default treatment to the remains nodes. Our approach in abstract interpretation share the same idea with example.

7.2 Compiling DSELs

Oege de Moor [5] described a technique for producing optimizing compilers for DSELs. The technique uses a data type of syntax for basic types, a set of smart constructors that perform rewriting over those types, some code motion transformations, and a back-end code generator. Domain-specific optimization results from chains of rewrites on basic types. This compiler is tailored for a specific application domain, i.e. those rewriting rules greatly exploited the optimization opportunities . This embedded optimizing compiler tends to remove the interpretative overhead which is caused by possible redundant computation. Our interpreter can separate the efficiency concern from the concern for user-friendliness and can also reduce the interpretative overhead by performing some optimization options, e.g. memoization.

Manuel Chakravarty [12] proposed another DSEL optimization technique: augment a language with compile-time meta-programming by using Template Haskell, which facilitates programmer in writing domain specific optimizations. The key idea of Manuel's approach is to represent code as a data structure, manipulate this data so that it represents equivalent but faster code, and finally turn this data back into code.

7.3 Monad

Category theory provides theoretical basis for the concept of monad[10, 9]. The monad used in Haskell [6] make it possible to implement imperative programming within pure functional language, such as IO monad and state monad. For the best of our knowledge, it is a novel view of monad as a software engineering tool to support the design of abstract data types.

8 Conclusion

The entire interpretation thus involves the following components:

- definition of a DSEL in a friendly syntax;
- introduction of analysis through overloading;
- definition of a virtual machine to encapsulate evaluation detail;
- existence of a main controller to coordinate the activities among the above components.

Let's discuss each of the above briefly.

The definition of DSEL will be in terms of combinators, rather than algebraic data types. The advantage is apparent; combinator-based definition provides a syntax that matches the actions in the corresponding application domain more aptly, and less artificially than use of algebraic data types.

The introduction of analysis poses the problem that occurs frequently in DSEL: the need for syntax analysis. Through the help of Haskell class system, we achieve the analysis through heavy use of overloading. In fact, we will demonstrate the encoding of abstract interpretation through overloading mechanism.

One feature of this construction of analysis is that it is domain-specific. That is, operations pertaining to the application domain are overloaded, but not those pertaining to the underlying host language (Haskell in this case). To do that, it is necessary to distinguish the domain-specific types from the underlying host types. This is accomplished in a fashion similar to the work done by Simon Peyton-Jones on "Scrapping the Boilerplate".

The definition of virtual machine serves to separate the efficiency concern from the concern for user-friendliness. Here, we show that combinators-based DSEL programs can be represented, if it is beneficial to do so, by concrete algebraic data types within the virtual machine. This enables more efficient execution. We define the virtual machine by yet another DSEL, whose combinators are used to support the front-end of the interpretation. This definition is inspired by the work of John Launchbury in defining a DSEL for graph traversal.

Finally, the main program of the interpreter takes in a DSEL program, performs the necessary analysis, and use it to drive the execution of the program at the virtual machine.

References

1. H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs, 2nd Edition*. MIT Press, 1996.
2. S. Anand, W.N. Chin, and S.C. Khoo. Charting patterns on price history. In *Proceedings of International Conference on Functional Programming*, pages 134–145, Florence, Italy, September 2001.
3. S. Anand, W.N. Chin, and S.C. Khoo. A lazy divide & conquer approach to constraint solving. In *Proceedings of the 14th IEEE International Conference On Tools with Artificial Intelligence*, pages 106–116, Washington DC, USA, November 2002.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.
5. C. Elliott, S. Finne, and O de Moor. Compiling embedded languages. Msr-tr-00-52, Microsoft Research, 2000.
6. Paul Hudak, John Peterson, and Joseph H. Fasel. A gentle introduction to haskell 98. 1999.
7. John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp Symb. Comput.*, 8(4):293–341, 1995.
8. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-specific languages*, pages 109–122. ACM Press, 1999.
9. Eugenio Moggi. An abstract view of programming languages. Ecs-lfcs-90-113, Edinburgh University, 1989.
10. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press, 1989.
11. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM Press, 1993.

12. Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising embedded dsls using template haskell. URL: http://www.cse.unsw.edu.au/~sseefried/papers.html, March 2004.