The Essence of AspectJ

[Functional Pearl]

Mark P. Jones
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
20000 NW Walker Road, Beaverton, Oregon 97006, USA
mpj@cse.ogi.edu

ABSTRACT

In the construction of a large software system, it is inevitable that some aspects of program behavior will cut across the structure of the code. The changes that are needed to support a new feature, for example, may be spread across several different points in the original program, making them harder to maintain and harder to reuse. The designers of "aspect-oriented" programming languages aim to tackle these problems by introducing new language mechanisms to capture cross-cutting concerns. Their hope is that this will allow different aspects of a program to be captured as independent entities that can be woven together automatically to produce a complete program.

In this paper, we focus on AspectJ, an aspect-oriented extension of Java that has been designed and implemented by a team at Xerox PARC to support an empirical assessment of aspect-oriented programming. The development of AspectJ has been driven largely by pragmatic concerns. Here, we provide a complimentary perspective by using interpreters, written in Haskell, to present a formal semantics for a simple aspect-oriented programming language, and so to distill the essence of AspectJ.

This paper provides a firm semantic foundation for the design of aspect-oriented programming languages. It illustrates the flexibility that can be obtained by writing interpreters in a monadic style, but also challenges us to contemplate how the ideas of aspect-oriented programming might be applied to the design of future functional languages.

1. INTRODUCTION

Our ability to build non-trivial software systems relies on composition techniques that enable us to construct complete programs from collections of smaller pieces. A combination of factors, most notably the application domain and the choice of implementation language, will typically lead developers to adopt a particular decomposition or program structure as a means of managing the complexity of a large system. Unfortunately, the same decomposition can also become a significant obstacle as a program evolves to meet the changing needs of its users. With luck, and by starting with good analysis and design, many of the modifications and extensions that are required will fit neatly into the structure of the original decomposition. But, inevitably, some will not, and will instead require invasive changes that cut across this structure. Thus, over a period of time, the code becomes a complex and tangled web of interacting, often interfering, and sometimes redundant parts. The end result is a fragile and unreliable program that is hard to understand, harder to modify, and impossible to maintain.

Many of these problems can be traced to the programming languages that are used to create software systems, each of which typically emphasizes a particular style of decomposition. For example, in a procedural or functional language, programs are often structured as a collection of modules, each of which contains a collection of type and function definitions. By contrast, collections of classes, each containing field and method definitions, are used to describe programs in typical object-oriented languages. Unfortunately, despite our best efforts, each of these decomposition mechanisms seems to support only certain kinds of program evolution, while other changes are, at best, not helped or, at worst, actively hindered. Thus, once the initial version of a program has been written, its subsequent evolution suffers from what Ossher and Tarr have aptly named "The Tyranny of the Dominant Decomposition" [9].

Aspect-Oriented Programming (AOP) [4] has been proposed as a way to address these problems by allowing programmers to abstract out different aspects of program behavior as independent, and potentially reusable components. In particular, the challenges in developing an aspect-oriented programming language are (i) in finding linguistic constructs that allow cross-cutting concerns to be captured in a modular fashion; and (ii) and in developing compilation techniques that allow different aspects to be combined (by a process that is sometimes referred to as *weaving*) to construct a complete application.

AspectJ [3] is a practical, aspect-oriented extension of Java [1] that has been developed by Gregor Kiczales and colleagues at Xerox PARC. The goals of the AspectJ project,

however, are much broader than the design and implementation of a new programming language: the hope is that AspectJ will provide a platform for empirical assessment of aspect-oriented programming. For example, it will be particularly interesting to see how programmers use the language to solve real problems, and to see whether it helps them to develop code that is more modular, more reusable, and easier to maintain and evolve. As such, the AspectJ team have focused on important pragmatic issues such as developing a robust and efficient compiler, adding support for popular integrated development environments (IDEs), and building and supporting a growing user community. With its focus on pragmatic concerns, AspectJ is not intended as a "pure" or "clean room" implementation of the concepts of AOP, nor as an attempt to explore the design space for AOP languages.

To date, the semantics of AspectJ have been described only by means of examples or informal explanations. In this paper, we take a more formal approach, using interpreters—written using the functional programming language Haskell—to provide a precise, executable semantics for aspect-oriented programming languages in the style of AspectJ. The language that our interpreters treat is a small, imperative language that is much simpler than Java or AspectJ: it is rich enough for us to describe the key ideas of AspectJ, while also being small enough to avoid the inevitable distractions of a larger set of language features. In this sense, and as the title of this paper suggests, we believe that our interpreters capture the essence of AspectJ.

The interpreters in this paper are written in Haskell 98 [10], henceforth referred to simply as just "Haskell", and have been tested using the Hugs interpreter [2] running in Haskell 98 mode. We do assume some degree of familiarity with functional programming, but we have also made an effort to explain some of the more intricate details of syntax and semantics that might not be familiar to readers with only limited experience of Haskell.

The remaining sections of this paper are as follows. In Section 2, we give a short introduction to the AspectJ language, focusing on the concepts of join points, pointcuts, advice, and aspects that it offers as tools for aspect-oriented programming. In Section 3, we give the syntax, and an executable semantics via a simple interpreter, for a small imperative language, Mini Pascal. This provides the starting point for our more formal treatment of aspect-oriented programming in later sections. A modified version of the semantics is presented in Section 4 by recasting the original interpreter in a monadic style. In Section 5, we describe how the key concepts of AspectJ can be mapped on to the simpler Mini Pascal setting to develop a simple aspect-oriented language, Aspect Pascal. Closing thoughts are presented in Section 6.

2. ASPECTJ

This section provides a brief overview of AspectJ, both to explain the mechanisms that it provides to support aspect-oriented programming, and to motivate the development in subsequent sections. As we have already indicated in the introduction, AspectJ is a practically-motivated, aspect-oriented language that is based on Java. Many Java programs are also perfectly valid AspectJ programs, and for much of the time, there is little to distinguish the experience of programming in one language from the other. The area in which the languages differ, of course, is in the facilities that AspectJ provides to support aspect-oriented programming. In the terminology of AspectJ, there are four important concepts: Join points, Pointcuts, Advice, and Aspects.

Join points are points in the dynamic execution graph of a program through which control flow passes. For example, in the context of the following Java class, a joint point might represent a particular call to the `incr()` method, or a specific assignment to the `n` field of some `Counter` object:

```
class Counter {
   private int n;
   void reset() { n = 0; }
   int  get()   { return n; }
   void incr()  { n = n+1; }
}
```

It is often useful, and perhaps even necessary, to distinguish between the times at which we enter and leave any given join point. For example, if a join point represents a call to `incr()`, then we would expect the value in the state variable `n` just before the function is invoked to be different from the value in `n` just after the function has returned. Anyone who has used a debugger will be familiar with the way that concepts like these are used to specify breakpoints or watch variables. Note, however, that there will not usually be a one-one correspondence between join points and points in the source code for a program. For example, there will not be any join points for parts of the program that do are not executed. At the same time, repeated execution of a particular section of the source program—in a loop or recursive function, for example—will generate multiple join points.

Pointcuts represent sets of join points. In general, it is hard to give meaningful names to individual join points, or to enumerate some particular set of join point that are of interest in a particular situation. And so, instead, AspectJ provides a simple algebra of pointcut expressions that can be used to build up the specification for a particular pointcut. Primitive pointcut expressions include things like `calls(void Counter.incr())`, which represents the set of all join points for calls to a `Counter` object's `incr()` method, or `sets(int Counter.n)`, which represents the set of all points at which the `n` field of a `Counter` object is assigned to. More complicated pointcuts can be described using Boolean operators. For example, the pointcut

```
sets(int Counter.n) || calls(void Counter.incr())
```

includes all the join points that correspond either to an assignment to an `n` field or to a call to the `incr()` method of a `Counter` object. Note that each call to an `incr()` method will result in two distinct join points in this particular pointcut: one for the call to `incr()` itself, and one for the assignment to `n` in the body of that method. The syntax of AspectJ allows pointcuts to be named:

```
pointcut setsOrIncs() :
  sets(int Counter.n) ||
  calls(void Counter.incr());
```

It is also possible for a pointcut to carry parameters, which can be used to capture information about the execution context in which a particular join point arose, such as the value of a parameter or the receiving object in a method call. We will not, however, consider these possibilities any further here.

The most important thing about pointcuts is that they give programmers a to refer to specific points in the execution of a program without having to modify its source code.

Advice is used to specify how the behavior of a program should be modified at the join points within a particular pointcut. For example, the following advice might be used to add a logging facility to `Counter` objects so that the time at which any counter is reset will be recorded in some global system log:

```
static before() : calls(void Counter.reset()) {
  resetLog.writeCurrentTime();
}
```

This is an example of `before` advice, specifying actions that are to be performed immediately before (i.e., on entry to) any of the join points in the specified pointcut (which, in this case, is just the set of join points corresponding to a `Counter` object's `incr()` method.) AspectJ supports a corresponding notion of `after` advice, as well as several other forms of advice that we will not discuss further here.

Aspects are modular, linguistic units that encapsulate crosscutting concerns. In the context of AspectJ, aspects declarations look much like standard Java class declarations except that they can also contain pointcut and advice declarations. For example, a modification of our original `Counter` program to support logging might be described by the following aspect declaration:

```
aspect Logging {
  pointcut logPlaces() :
     calls(void Counter.reset());

  static before() : logPlaces() {
     resetLog.writeCurrentTime();
  }

  // ... code to define an initialize
  // the resetLog variable goes here ...
}
```

This example gives a taste of the kind of enhanced modularity that can be obtained using AspectJ:

- The original `Counter` program does not need to be modified at all to support the additional functionality of Logging.
- The `Counter` program can be compiled either with or without the `Logging` aspect to obtain whichever functionality is needed for a particular application; and
- All of the parts of the program related to logging are collected together in one place, resulting in code that is easier to maintain and evolve. For example, we can add support for logging of calls to different functions by changing the definition of the `logPlaces` pointcut, and we can specify more sophisticated logging behavior by modifying the `before` advice in the `Logging` aspect; neither of these modifications would require changes in other parts of the program.

On the other hand, it should also be quite clear that the mechanisms we have seen here are very powerful, and could easily be misused to construct programs that are much harder to understand and maintain than equivalent programs written in a conventional language. Of course, such problems are not unique to AspectJ: any language that gives programmers the flexibility to create interesting applications is also likely to offer many opportunities for create incomprehensible, unmaintainable code. The problems that motivate the development of aspect-oriented languages such as AspectJ are real, and they are not restricted to object-oriented languages—as developers working on any large and evolving functional program would probably confirm! In the following sections of this paper, we describe the beginnings of a process that we hope will lead to more formal semantic foundations for languages like AspectJ. In turn, we hope that this will result in a better understanding of the strengths and limitations of such languages as well as a clearer picture of the wider design space for aspect-oriented languages.

3. MINI PASCAL

In this section, we introduce a simple, imperative language that will be referred to in the following as "Mini Pascal." The syntax and semantics of this language are, by design, straightforward and, we hope, uncontroversial; this will allow us to avoid unnecessary distractions when we use the language as a platform for the aspect-oriented extensions that are described in later sections.

3.1 A Syntax for Mini Pascal

The abstract syntax of Mini Pascal is captured by the type definitions in Figure 1. There are four syntactic categories, represented by the types *Id* (identifiers), *IExp* (integer-valued expressions), *BExp* (Boolean-valued expressions), and *Stmt* (Statements). The language of integer-valued expressions includes literals (represented by *Lit*), the four basic arithmetic operators (represented by `:+:`, `:-:`, `:*:`, and `:/:`) and variables (represented by *Var*). The language of Boolean-valued expressions includes constants (T and F), the standard complement of Boolean operators (negation, conjunction, and disjunction, represented by *Not*, `:&:`, and `:|:`, respectively), and two integer comparison operators (`:=:` and `:<:`). For simplicity, Mini Pascal does not include any Boolean-valued variables. Finally, the language of statements includes the empty statement (*Skip*), assignments (written using :=), blocks (*Begin*), conditionals (*If*), and while loops (*While*).

```
type Id     =  String

data IExp   =  Lit Int
            |  IExp :+: IExp
            |  IExp :*: IExp
            |  IExp :-: IExp
            |  IExp :/: IExp
            |  Var Id

data BExp   =  T
            |  F
            |  Not BExp
            |  BExp :&: BExp
            |  BExp :|: BExp
            |  IExp :=: IExp
            |  IExp :<: IExp

data Stmt   =  Skip
            |  Id := IExp
            |  Begin [Stmt]
            |  If BExp Stmt Stmt
            |  While BExp Stmt
```

Figure 1: Abstract syntax for Mini Pascal

We have made significant use of infix operators here to increase the readability of our abstract syntax. The leading colon in many of the operator names is required by Haskell to signal the use of an infix operator as a data constructor, but it also helps to distinguish the operators from their obvious counterparts in Haskell.

The following program illustrates one possible concrete syntax for Mini Pascal in a program that calculates the sum of the numbers from 1 to 10:

```
total := 0;
count := 0;
while count < 10 do
begin
  count := count + 1;
  total := total + count
end
```

Using the abstract syntax described here, the same program can be represented by the following expression:

```
program  =
  Begin [
    "total" := Lit 0,
    "count" := Lit 0,
    While (Var "count" :<: Lit 10)
      (Begin [
          "count" := (Var "count" :+: Lit 1),
          "total" := (Var "total" :+: Var "count")
      ])
  ]
```

We will use this particular program throughout the paper to illustrate the effects of applying different aspect extensions.

Clearly, Mini Pascal omits many important features that we would expect to find in a more realistic imperative language like Pascal, and all of the object-oriented extensions that are included in languages like Java and AspectJ. For example, Mini Pascal has only two types of value, one type of variable, and no support for functions or methods. Even so, we will see that it is still rich enough to be useful in illustrating the key ideas of AspectJ.

3.2 A Direct Semantics for Mini Pascal

The meaning of Mini Pascal programs is described by semantic functions that interpret each of the syntactic categories of the language in an appropriate way. Of course, the assignment statement and variable lookup expression of Mini Pascal operate on an implicit store that must be made explicit in our semantics. We will model these stores as functions mapping identifiers to corresponding integers:

```
type Store    =  Id → Int

extend        ::  Store → Id → Int → Store
extend s i v  =  \j → if i == j then v else s j
```

The *extend* function defined here will be used to describe the process of updating a store: *extend s i v* represents the store that is obtained from s by binding the value v to the identifier i. (This example also shows the syntax that Haskell uses for λ-expressions or anonymous functions: the expression $\backslash x \rightarrow e$ denotes the function that maps an input parameter x to the corresponding value of expression e.) We will assume that programs begin execution in a store where all variables are initialized to zero. In Haskell, this can be described succinctly by the expression (*const* 0); the *const* function used here is the name that is used for the standard K combinator in the Haskell prelude, and is defined by the equation $const\ x\ y = x$.

Now we can complete the semantics for Mini Pascal, using a semantic function *iexp* to interpret each integer-valued expression as a function from stores to integers; a semantic function *bexp* to interpret each Boolean-valued expression as a function from stores to Booleans; and a semantic function *stmt* to interpret each statement as a store transformer:

```
iexp  ::  IExp → (Store → Int)
bexp  ::  BExp → (Store → Bool)
stmt  ::  Stmt → (Store → Store)
```

The definitions for each of these functions are shown in Figure 2, and will not hold any surprises. In most cases, the constructs of Mini Pascal are interpreted by the corresponding constructs of Haskell. For example, the `:+:` operator is interpreted by the addition operator (+) on values of type *Int*, while the Boolean conjunction `:&&:` is interpreted by the standard Haskell conjunction operator `&&`. The interpretation of blocks in the definition for *stmt* uses the *foldr* function, which may not be familiar to those with limited experience of Haskell. In this particular case, it is used simply to give a concise definition of the semantics of a block—which is just a sequence of statements, written using the *Block* constructor—as the composition of the transformations for each statement in the block. (In Haskell, an infix period (.) is used to denote function composition.) Informally, the semantics of blocks can be explained by the fol-

```
iexp                 ::  IExp → (Store → Int)
iexp (Lit n)         =   const n
iexp (e1 :+: e2)     =   \s → iexp e1 s + iexp e2 s
iexp (e1 :*: e2)     =   \s → iexp e1 s * iexp e2 s
iexp (e1 :-: e2)     =   \s → iexp e1 s - iexp e2 s
iexp (e1 :/: e2)     =   \s → iexp e1 s `div` iexp e2 s
iexp (Var i)         =   \s → s i

bexp                 ::  BExp → (Store → Bool)
bexp T               =   const True
bexp F               =   const False
bexp (Not b)         =   \s → not (bexp b s)
bexp (b1 :&: b2)     =   \s → bexp b1 s && bexp b2 s
bexp (b1 :|: b2)     =   \s → bexp b1 s || bexp b2 s
bexp (e1 :=: e2)     =   \s → iexp e1 s == iexp e2 s
bexp (e1 :<: e2)     =   \s → iexp e1 s < iexp e2 s

stmt                 ::  Stmt → (Store → Store)
stmt Skip            =   \s → s
stmt (i := e)        =   \s → extend s i (iexp e s)
stmt (Begin ss)      =   foldr (\s ss → ss . stmt s) id ss
stmt (If b t e)      =   \s → if bexp b s
                              then stmt t s
                              else stmt e s
stmt (While b t)     =   loop
  where loop s       =   if bexp b s
                             then loop (stmt t s)
                             else s
```

Figure 2: A direct-style interpreter for Mini Pascal

lowing equation:

$$stmt\ (Block\ [s_1, \ldots, s_n]) \quad = \quad stmt\ s_n\ .\quad \ldots\quad .\ stmt\ s_1$$

Notice that neither *iexp* or *bexp* returns a modified store, reflecting the fact that Mini Pascal expressions do not have any side-effects: there is no way for the evaluation of an expression to change the store.

We can use the functions described here to build a simple test harness for executing Mini Pascal programs, producing the final store as a result:

```
run      ::  Stmt → Store
run s    =   stmt s (const 0)
```

Combining this with a simple function *demo* that displays the values of the variables "`count`", "`total`", and "`other`" in the final store, we can see the result of executing the example *program* from Section 3.1[1]:

```
Hugs> demo (run program)
count = 10
total = 55
other = 0

Hugs>
```

[1]The text `Hugs>` in the output shown here is just the prompt of the Hugs interpreter that was used to run the examples in this paper.

The final values shown for "`count`" and "`total`" are exactly what we would expect for this program. The value shown for "`other`" is also correct; this particular variable is not used in the sample program and so the final value is just zero, which was the value assigned to the variable in the initial store, *const* 0.

4. A MONADIC INTERPRETER

As a first step towards an aspect-oriented version of Mini Pascal, we will recast the interpreter from Figure 2 in a monadic style. Monads were originally studied quite extensively in abstract areas of mathematics such as category theory and universal algebra. More recently, monads have received attention for the role that they can played in extending purely functional languages with support for side-effecting features such as I/O [11] and state [5]. In this paper, however, our focus is on the use of monads in denotational semantics, as described by Moggi [7, 8], and in structuring functional programs, as described by Wadler [12]. In fact, as many readers will have already realized, the title of the present paper is, in part, a homage to Wadler's "Essence of Functional Programming," which showed how monads could be used to present several variations of an interpreter for the λ-calculus, and to which we refer any reader who finds that the following introduction to monadic programming is too brief!

From a programming perspective, we can think of a monad as a particular kind of abstract datatype whose values are used to model computations. In the context of Mini Pascal, the computations that we are interested in are state transformers: functions that map a store to a value of some result type r, and also return a (possibly updated) store:

$$\textbf{data}\ M\ r \quad = \quad ST\ (Store \to (r,\ Store))$$

Given this datatype, we can use the constructor function ST to wrap up useful state transformers as values with types of the form $M\ r$:

```
setVar       ::  Id → Int → M ()
setVar i v   =   ST (\s → ((), \j → if i == j then v
                                               else s j))

getVar       ::  Id → M Int
getVar i     =   ST (\s → (s i, s))
```

A computation $setVar\ i\ v$ describes the state transformer that updates the initial store with a new value v for variable i, and returns the unit value (). In this case, the result type r is just the unit type, (), and so the type of the computation $setVar\ i\ v$ is just $M\ ()$. A $getVar\ i$ computation, on the other hand, returns the integer value for the variable i in the initial store, and does not modify the store component. Hence the result type r in this case is Int, and the type of the computation itself is $M\ Int$.

To complete the definition of a monad, we must define two special operators called *return* and bind (written as an infix operator, `>>=`). The following declaration signals that M is indeed a monad by providing appropriate definitions for

these functions:

```
instance Monad M where
  return x  =  ST (\s -> (x, s))
  c >>= g   =  ST (\s -> let ST f  =  c
                             (x, s') =  f s
                             ST f'   =  g x
                         in f' s')
```

From the definition of *return*, we can see that a computation of the form *return x* is a state transformer that just returns the value x without modifying the state. The bind operator corresponds to a sequencing of computations. To execute a computation of the form c `>>=` g, we run the computation c on the initial state s to obtain a result x and a modified state s'. Then we apply g to x to obtain a new computation that we can execute with initial state s' to produce the final result.

In this paper, we will not use the `>>=` operator directly. Instead, we will make use of Haskell's **do**-notation, which the compiler translates automatically into corresponding expressions involving `>>=`. For example, the following function definition uses this syntax[2]:

```
liftM2           :: (a -> b -> c) -> M a -> M b -> M c
liftM2 op c d    =  do x <- c
                       y <- d
                       return (op x y)
```

This definition has a simple, and intuitive reading: run the computation c to obtain a result x; then run the computation d to obtain a result y; finally, use the operator *op* to combine the two values x and y and return this value as the final result. However, a Haskell compiler will actually implement the *liftM2* function by translating the previous definition into the following code:

```
liftM2 op c d
    =  c >>= (\x -> d >>= (\y -> return (op x y)))
```

Whichever definition of *liftM2* you might prefer, the important thing to notice in each case is that there is *no explicit mention of the underlying store component*; the monad operators that we have defined take care of propagating the store from one statement to the next so that the programmer doesn't need to do this explicitly.

With these monadic preliminaries behind us, the monadic version of our interpreter, which is shown in Figure 3, should be reasonably straightforward. In this version of the interpreter, *IExp* values are mapped into computations that return integers (i.e., into values of type *M Int*); *BExp* value are mapped into computations that return Booleans (i.e., into values of type *M Bool*); and statements, which are executed only for their effect on the store, are mapped into computations that return values of unit type (i.e., into values of type *M* ()). Some readers will have noticed that this approach is more general than is strictly necessary: as we have already indicated, there is no way for an expression to change the store, and hence it is not necessary for us to interpret *IExp* or *BExp* values using the monad M. We will see, however, that this extra flexibility is important when we extend our interpreter again in the next section to support aspects.

[2]We have chosen the strangely named *liftM2* function as our example here because it will play an important role in giving the semantics to binary operators in our monadic operator. In fact this operator is actually defined, using the same name, in the standard Haskell library called *Monad*.

```
iexp               :: IExp -> M Int
iexp (Lit n)       =  return n
iexp (e1 :+: e2)   =  liftM2 (+) (iexp e1) (iexp e2)
iexp (e1 :*: e2)   =  liftM2 (*) (iexp e1) (iexp e2)
iexp (e1 :-: e2)   =  liftM2 (*) (iexp e1) (iexp e2)
iexp (e1 :/: e2)   =  liftM2 (*) (iexp e1) (iexp e2)
iexp (Var i)       =  getVar i

bexp               :: BExp -> M Bool
bexp T             =  return True
bexp F             =  return False
bexp (Not b)       =  liftM not (bexp b)
bexp (b1 :&: b2)   =  liftM2 (&&) (bexp b1) (bexp b2)
bexp (b1 :|: b2)   =  liftM2 (||) (bexp b1) (bexp b2)
bexp (e1 :=: e2)   =  liftM2 (==) (iexp e1) (iexp e2)
bexp (e1 :<: e2)   =  liftM2 (<) (iexp e1) (iexp e2)

stmt               :: Stmt -> M ()
stmt Skip          =  return ()
stmt (i := e)      =  do x <- iexp e; setVar i x
stmt (Begin ss)    =  mapM_ stmt ss
stmt (If b t f)    =  do x <- bexp b
                         if x then stmt t
                              else stmt f
stmt (While b t)   =  loop
  where loop       =  do x <- bexp b
                         if x then (stmt t >> loop)
                              else return ()
```

Figure 3: A monadic interpreter for Mini Pascal

To test our monadic interpreter, we will define a new version of the *run* operator. This function uses the *stmt* function to calculate the store transformer *st* corresponding to its argument s. The effect of running s is just the store value that is produced by applying *st* is applied to the initial store, (*const* 0):

```
run     :: Stmt -> Store
run s   =  snd (st (const 0))
           where ST st  =  stmt s
```

Although we have changed the definition of the interpreter in quite significant ways, it still behaves in the same way as the original version that we saw in Section 3.2:

```
Hugs> demo (run program)
count = 10
total = 55
other = 0

Hugs>
```

5. ASPECT PASCAL

In this section, we will show how our monadic interpreter for Mini Pascal can be modified to obtain an interpreter (in

Section 5.4) for Aspect Pascal, a simple aspect-oriented language in the style of AspectJ. In particular, we will describe Aspect Pascal in terms of suitable notions of join points (Section 5.1), pointcuts (Section 5.2) and advice (Section 5.3).

5.1 Join Points in Mini Pascal

Our first task is to decide what we mean by a joint point in a Mini Pascal program. More specifically, we need to identify the points at which the behavior of a program might usefully be modified or extended by subsequent advice. For the purposes of this paper, we will use just two types of join point—corresponding to the places at which the value of a variable is read or set—and we will use values of the following datatype to describe each join point:

$$\textbf{data}\ \mathit{JoinPointDesc} \;=\; \mathit{Get}\ \mathit{Id} \mid \mathit{Set}\ \mathit{Id}$$

The intention here is that a value *Get i* will be used to describe a join point at which the variable i is read, while a value *Set i* will be used to describe a join point at which a value is assigned to the variable i. Note that values of type *JoinPointDesc* represent descriptions of join points, and many different join points may be mapped to the same description value.

5.2 Pointcuts in Mini Pascal

As in AspectJ, we will use a language of pointcut expressions to describe sets of join points. The abstract syntax that we will use for pointcuts in Aspect Pascal is as follows:

$$\begin{array}{lcl}\textbf{data}\ \mathit{Pointcut} & = & \mathit{Setter}\\ & \mid & \mathit{Getter}\\ & \mid & \mathit{AtVar}\ \mathit{Id}\\ & \mid & \mathit{NotAt}\ \mathit{Pointcut}\\ & \mid & \mathit{Pointcut}\ \texttt{:||:}\ \mathit{Pointcut}\\ & \mid & \mathit{Pointcut}\ \texttt{:\&\&:}\ \mathit{Pointcut}\end{array}$$

Here, we have three primitive pointcut expressions:

- *Setter* represents the pointcut of all join points at which the value of a variable is being set.
- *Getter* represents the pointcut of all join points at which the value of a variable is being read.
- *AtVar i* represents the point cut of all join points at which the value of a the variable i is being set or read.

There are three additional *Pointcut* constructors, corresponding to Boolean negation (*NotAt*), disjunction ((`:||:`)), and conjunction (`:&&:`), that can be used to build up more complex pointcuts expressions. For example, the pointcut of all join points at which the variable "`x`" is set can be described by the expression (*Setter* `:&&:` *AtVar* "`x`"). More formally, we can describe the semantics of pointcuts as sets of join points (represented by characteristic functions of type *JoinPointDesc* → *Bool*) by using the following function:

$$\begin{array}{lcl}\mathit{includes} & :: & \mathit{Pointcut} \to (\mathit{JoinPointDesc} \to \mathit{Bool})\\ \mathit{includes}\ \mathit{Setter}\ (\mathit{Set}\ i) & = & \mathit{True}\\ \mathit{includes}\ \mathit{Getter}\ (\mathit{Get}\ i) & = & \mathit{True}\\ \mathit{includes}\ (\mathit{AtVar}\ i)\ (\mathit{Get}\ j) & = & i == j\\ \mathit{includes}\ (\mathit{AtVar}\ i)\ (\mathit{Set}\ j) & = & i == j\\ \mathit{includes}\ (\mathit{NotAt}\ p)\ d & = & \mathit{not}\ (\mathit{includes}\ p\ d)\\ \mathit{includes}\ (p\ \texttt{:||:}\ q)\ d & = & \mathit{includes}\ p\ d\ \texttt{||}\ \mathit{includes}\ q\ d\\ \mathit{includes}\ (p\ \texttt{:\&\&:}\ q)\ d & = & \mathit{includes}\ p\ d\ \texttt{\&\&}\ \mathit{includes}\ q\ d\\ \mathit{includes}\ \_\ \_ & = & \mathit{False}\end{array}$$

5.3 Advice in Mini Pascal

In Aspect Pascal, as in AspectJ, modifications to a program are described using a notion of advice. Each piece of advice includes a pointcut to specify the join points at which the advice should be used, and an arbitrary statement, to specify the action that should be performed. We will distinguish between two kinds of advice here: *Before* advice, which will be executed on entry to a join point, and *After* advice, which will be executed on the exit from a join point:

$$\begin{array}{lcl}\textbf{data}\ \mathit{Advice} & = & \mathit{Before}\ \mathit{Pointcut}\ \mathit{Stmt}\\ & \mid & \mathit{After}\ \mathit{Pointcut}\ \mathit{Stmt}\end{array}$$

The following definitions provide several examples of advice that might be used in combination with the example *program* from Section 3.1:

$$\begin{array}{l}\mathit{badCountSets}\\ \quad = \ \mathit{After}\ \mathit{Setter}\\ \qquad\quad (\text{“}\texttt{other}\text{”} := (\mathit{Var}\ \text{“}\texttt{other}\text{”}\ \texttt{:+:}\ \mathit{Lit}\ 1))\end{array}$$

$$\begin{array}{l}\mathit{countSets}\\ \quad = \ \mathit{After}\ (\mathit{Setter}\ \texttt{:\&\&:}\ (\mathit{NotAt}\ (\mathit{AtVar}\ \text{“}\texttt{other}\text{”})))\\ \qquad\quad (\text{“}\texttt{other}\text{”} := (\mathit{Var}\ \text{“}\texttt{other}\text{”}\ \texttt{:+:}\ \mathit{Lit}\ 1))\end{array}$$

The *badCountSets* example here shows one attempt to maintain a count of the total number of assignments that are performed during the execution of a program. It indicates that the "`other`" variable should be incremented each time we reach a *Setter* join point. With the semantics of Aspect Pascal that will be presented in Section 5.4, this example will not work properly. Our interpreter treats every part of a program with the same advice, including those parts that are themselves the result of advice. Hence the attempt to increment "`other`" when a *Setter* join point is encountered will result in another *Setter* join point, and a recursive invocation of the associated advice, resulting in a non-terminating computation. The definition of *countSets* shows how this can be remedied by using a narrower pointcut that excludes join points for the "`other`" variable, but includes all other *Setter* join points. Instead of leaving it to the programmer to detect and deal with subtleties like this, it would also be possible to change the language design. For example, we could modify the semantics of Aspect Pascal to avoid recursive application of advice altogether. Alternatively, we might arrange for the statement component of an advice value to be executed only at join points that are included in the associated pointcut, and which do not involve any of the variables used in the statement. These alternatives suggest

that there are a range of possible language design choices, but we will not pursue them any further in this paper.

The following definition of *countGets* uses the same trick as *countSets* to provide an aspect that will count the number of variable reads during the execution of a program:

```
countGets
    = After (Getter :&&: (NotAt (AtVar "other")))
            ("other" := (Var "other" :+: Lit 1))
```

Examples like this suggest that aspects could be useful as a tool for certain forms of profiling. Our final example in this section shows how advice might be used to modify the behavior of an assignment to the variable "`total`" so that its previous value is recorded in the variable "`other`". In this case, the use of *Before* advice, rather than *After* advice, is essential to obtain the intended behavior:

```
saveLastTotal
    = Before (Setter :&&: AtVar "total")
             ("other" := Var "total")
```

In the terminology of AspectJ, any combination of advice values can be considered as an aspect, and so we will use lists of *Advice* values to represent aspects in Aspect Pascal:

```
type Aspects = [Advice]
```

For example, if we run our example *program* with the empty list of aspects, [], then we would expect the same results that we saw with the interpreters in Sections 3.2 and 4. If, on the other hand, we used the combination of aspects [*countGets*, *countSets*], we would expect to obtain a program that reports the total number of reads and assignment in the variable "`other`".

5.4 A Semantics for Aspect Pascal

A complete Aspect Pascal program consists of two components: a list of aspects and a single Mini Pascal statement to which those aspects should be applied. We can reflect this by extending the *run* function of previous interpreters to take an extra argument of type *Aspects*:

```
run       :: Aspects → Stmt → Store
run a s   =  snd (st a (const 0)) where ST st  =  stmt s
```

This, in turn, requires a change to our monad M to ensure that the initial list of aspects is propagated to each part of the program during execution; in other words, each of our store transformers must now be modified parameterized by a list of aspects. The modified definitions for M, and for the monad operators *return* and `>>=`, are as follows:

```
data M a  =  ST (Aspects → Store → (a, Store))

instance Monad M where
  return x  =  ST (\a s → (x, s))
  c >>= g   =  ST (\a s → let ST f   =  c
                              (x, s') =  f a s
                              ST f'   =  g x
                          in f' a s')
```

It is also quite easy to modify our previous definitions of *setVar* and *getVar* to allow for the presence of the new *Aspects* parameter:

```
setVar       :: Id → Int → M ()
setVar i v   =  ST (\a s → ((), \j → if i == j then v
                                     else s j))

getVar       :: Id → M Int
getVar i     =  ST (\a s → (s i, s))
```

(As an aside, readers with experience using monads may recognize that we what we are doing here just corresponds to applying an environment or reader monad transformer, and lifting the operations on *Store* values in an appropriate way [6].) An additional monad operator is required to allow access to the list of *Aspects* at any point during execution:

```
getAspects  :: M Aspects
getAspects  =  ST (\a s → (a, s))
```

We will use the *getAspect* operator only once, but in a critical way, to define an operator *withAdvice* that makes the link between program execution and aspects using join points. More precisely, if c is a computation corresponding to some join point with description d, then *withAdvice d c* wraps the execution of c with the execution of the appropriate *Before* and *After* advice, if any:

```
withAdvice       :: JoinPointDesc → M a → M a
withAdvice d c   =  do aspects ← getAspects
                       mapM_ stmt (before d aspects)
                       x ← c
                       mapM_ stmt (after d aspects)
                       return x
```

This function has a natural reading: read the list of *aspects* that are required for this execution of the program; execute any *before* aspects corresponding to the join point described by d; execute the computation c to obtain a result x; execute any *after* aspects corresponding to the join point described by d; and, finally, return the value of x as the result of the whole computation. The *mapM_ stmt* function used here sequences the execution of a list of statements (the *mapM_* operator is defined in the standard Haskell prelude), while *before* and *after* are simple utility functions that select the before and after advice for a given join point description d from a list of aspects *as*:

```
before, after  :: JoinPointDesc → Aspects → [Stmt]
before d as    =  [s | Before c s ← as, includes c d]
after d as     =  [s | After c s ← as, includes c d]
```

To complete our description of the semantics of Aspect Pascal, we need to modify the way that expressions and statements are interpreted. In most cases, we can use exactly the same definitions that were given in Figure 3. The only places where changes are required, as shown in Figure 4, are in the interpretation of variable expressions and assignment statements, where the original calls to *getVar* and *setVar* are wrapped by applications of *withAdvice* together with the appropriate join point descriptions.

Now, at last, we can test our interpreter by running the sample *program* from Section 3.1 with different combinations of the advice values from Section 5.3:

```
Hugs> demo (run [] program)
```

```
iexp              ::  IExp → M Int
                  ... = ...
iexp (Var i)      =   withAdvice (Get i) (getVar i)

stmt              ::  Stmt → M ()
                  ... = ...
stmt (i := e)     =   do x ← iexp e
                         withAdvice (Set i) (setVar i x)
```

Figure 4: An interpreter for Aspect Pascal

```
count = 10
total = 55
other = 0

Hugs> demo (run [countSets] program)
count = 10
total = 55
other = 22

Hugs> demo (run [countGets] program)
count = 10
total = 55
other = 41

Hugs> demo (run [countGets, countGets] program)
count = 10
total = 55
other = 43

Hugs> demo (run [saveLastTotal] program)
count = 10
total = 55
other = 45

Hugs>
```

6. CONCLUSIONS

Work on aspect-oriented programming deals with the difficult and real problems that occur when we try to describe modifications to a program that cut across the program's structure. AspectJ represents one point in the design space for aspect-oriented programming languages, and provides, in a Java compatible package, a solid, and well-worked out set of AOP features. In this paper, we have used interpreters, written in Haskell, to provided a formal, but also executable semantics for a simple aspect-oriented language in the style of AspectJ. Our hope is that this work will lead to a deeper understanding of both AspectJ itself, and of the broader design space.

It is clear that there are many possible ways to generalize our interpreter to deal with other kinds of join point. For example, we could modify the definition of join point descriptions to include extra cases, such as:

```
data JoinPointDesc = ...
                   | GetVal Int
                   | SetVal Int Int
                   | FDiv
```

Here, *GetVal n* might describe a join point at which a value n is being read from a variable; *SetVal new old* could describe a join point at which an *old* value is being replaced by a *new* value in a variable; and *FDiv* might be used to describe join points at which a division operation is performed. (The latter might be useful, for example, on a computer where the corrective action might sometimes be needed to work around bugs in a particular CPU's implementation of division!) Occurrences of corresponding join points during program execution can easily be flagged by wrapping the appropriate lines in the definition of our interpreter with suitable calls to the *withAdvice* function.

It is also possible to generalize our language of pointcuts to include new forms of expression such as:

```
data Pointcut = ...
              | WithVal (Int → Bool)
              | AtVarMatch (Id → Bool)
              | InStore (Store → Bool)
```

Each of the additions shown here provides a predicate that might be used to capture arbitrary conditions on the value that is being read from a variable (using *WithVal*), or on the name of the variable that is being accessed (using *AtVarMatch*), or on the store in which the join point occurs (using *InStore*). These examples illustrate the potential for generalization and extensions; of course, implementing some of these features would carry a significant overhead.

In fact, the current AspectJ implementation is not based on an interpreter, but instead uses a program transformation called *weaving*. In the context of this paper, we might try to describe this implementation strategy by a function:

```
weave :: Aspects → Stmt → Stmt
```

In fact, it is not clear whether the abstract syntax for Mini Pascal is rich enough for us to define such a function, but it could certainly be extended to allow experimentation and formalization of this approach.

As a final comment, in the sequence of interpreters that we have shown here, we have actually demonstrated exactly the kinds of problems that motivate the need for aspect-oriented programming. Neither the change from our original interpreter to the monadic version, nor the subsequent evolution of that program to an interpreter for Aspect Pascal were neatly encapsulated as reusable, modular units of change! Some readers may suggest that this is the result of a poor design in the original program. I hope, however, that many readers will consider this as a clear indication of the potential for an aspect-oriented functional language!

7. ACKNOWLEDGMENTS

The original versions of the interpreters described in this paper were written after an informative and insightful tutorial on AspectJ that was presented by Gregor Kiczales at a meeting for the DARPA project on "Program Composition for Embedded Systems." The work has also benefited considerably from feedback provided by members of the Pacific Software Research Center (PacSoft) at OGI, and particularly from stimulating and ongoing discussions with Andrew Black on the subject of aspect-oriented programming and related topics.

8. REFERENCES

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison Wesley, 2000.

[2] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs/`.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. Draft white paper, submitted for publication. Available from `http://www.aspectj.org`.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag LNCS 1241.

[5] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[6] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 1995.

[7] E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, 1989.

[8] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[9] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. Research Report 21452, IBM, April 1999.

[10] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

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

[12] P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.