Adding Superimposition To a Language Semantics
— Extended Abstract —

Ralf Lämmel[1,2]
[1] CWI, Kruislaan 413, NL-1098 SJ Amsterdam
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Ralf.Laemmel@cwi.nl

Abstract

Given the denotational semantics of a programming language, we describe a general method to extend the language in a way that it supports a form of *superimposition* — just in the sense of aspect-oriented programming. In the extended language, the programmer can superimpose additional or alternative functionality (aka advice) onto points along the execution of a program. Adding superimposition to a language semantics comes down to three steps: (i) the semantic functions are elaborated to carry advice; (ii) the semantic equations are turned into 'reflective' style so that they can be altered at will; (iii) a construct for binding advice is integrated. We illustrate the approach by representing semantics definitions as interpreters in Haskell.

1 Introduction

One might say that the essence of an aspect-oriented programming (AOP) language like AspectJ is that it is an amalgamated language in the following sense. Besides ordinary object-oriented expressiveness, one can also write code that superimposes advice onto points along the execution of object-oriented functionality. An important class of join points are method calls. One can give a precise definition of this sort of AOP on the basis of a formal semantics [8]. (There are further models of AOP, e.g., the Hyper/J-like model, which we will not address in this paper.)

Question: Can the perception of 'superimposition' be captured in a language-parametric manner, that is, without talking about method calls, or other constructs, without commitment to a specific language? There has been work on studying some forms of superimposition (say, AOP) at a fundamental level [3, 6, 1, 4] on the basis of specific computational models. However, we seek an approach that applies immediately to actual programming languages.

In this extended abstract, we describe a general method to add superimposition to the denotational semantics of a language. The overall approach is described in Sec. 2. An illustrative example is developed in full detail in Sec. 3. The design space for our form of superimposition is briefly scanned in Sec. 4. Related work is discussed in Sec. 5, and the paper is concluded in Sec. 6.

2 The overall approach

Suppose we consider a semantic function of the following type:

$$[\![\cdot]\!] : S_i \to D_i$$

Here, S_i is a syntactical domain, and D_i is the associated domain of semantic meanings, say *denotations*. Semantic functions are specified using case discrimination on S_i while the meaning of a syntactical form is expressed in terms of the meanings of its subterms (aka compositionality). Let us consider a specific semantic equation. Without loss of generality, we pick up an equation for a construct with one subconstruct:

$$[\![C\ x]\!] = f\ [\![x]\!]$$

Here, $C\ x$ is the syntactical pattern at hand, and f is an operation that turns the denotation of x into the denotation of $C\ x$. (We use curried function application.) Adding superimposition is now performed in three steps:

1. The semantic functions are elaborated to carry advice.
2. Denotations are made 'reflective' so that they can be altered.
3. A construct for binding advice is integrated.

The type of the above semantic function is adapted as follows:

$$[\![\cdot]\!] \quad : \quad S_i \to R\ D_i$$

Here, R is a domain constructor to add a *registry* with superimposed advice to domains of ordinary denotations. There are a few options for the actual type of the registry. In essence, the registry is a transformer for reflective denotations. To obtain a modular semantics, we assume that R is a monad. Depending on the binding policy and other language-design decisions, R could be the environment monad, the state monad, or a combination of both.

The semantic equation from above is adapted as follows:

$$[\![C\ x]\!] \quad = \quad (C\ x) \mathrel{\textbf{:-}} ([\![x]\!] \gg\!= \lambda x'.\ \mathit{return}\ (f\ x'))$$

The occurrences of "$\gg\!=$" and *return* point to monadic style. (We use the common convention that the monad operator, which lifts values to computations, is denoted by *return*, whereas the monad operator, which applies a value consumer to a computation, is denoted by "$\gg\!=$".) The semantic equations can be turned into monadic style by a systematic transformation. It is avoidable to introduce monadic style if we fix the binding policy for advice, e.g., by explicitly passing around an environment for the registry.

The occurrence of the infix operator " **:-** " in the above equation points to reflective style. Both the operator " **:-** " and this use of the term 'reflective style' are inventions of the present paper. The operator " **:-** " shapes the reflective denotation as follows. If d is the original denotation for a term t, and d_m is d in monadic style, then the reflective denotation is of the form $t \mathrel{\textbf{:-}} d_m$. This is read as 'by default, at t do d_m'. There are a few options for the precise definition of " **:-** ". In essence, the operator does nothing but application of denotation transformers as provided by the registry. That is, given the registry r, the term t, and the denotation d, the operation t **:-** d applies r to t and d. So basically, each semantic equation can be revised via the registry r.

It remains to perform the third step from above, that is, the integration of a construct for binding advice. This comes down to adding one equation for $[\![\cdot]\!]$. There are a few options depending on the favoured binding policy and other language-design decisions. One option is to hold superimposed advice in an environment with a binding scope that is local to a given program fragment. That is:

$$[\![h \ \textbf{adapts} \ t]\!] \quad = \lambda h'.\ [\![t]\!]\ h$$

We call h and h' *hooks*. The old hook h' is replaced by the new hook h. We say that h *adapts* the program fragment t. The denotation transformers accomplished by h will transform the denotations determined for t and its subconstructs. (We try not to use the AOP term 'advice' for h because h accomplishes both the advice code and the definition of join points or pointcuts.) Hooks and registries are of the same type: a family of denotation transformers indexed by the syntactical domains. For example, the denotation transformer for S_i is of type $S_i \rightarrow R\ D_i \ \rightarrow \ R\ D_i$. The first argument of type S_i emphasises that reflective denotation transformers can look at the program text to make a decision.

Our method works, in principle, for all possible language semantics (such as imperative languages, different object-oriented languages, functional languages, etc.). However, language-specific properties and obligations occur in this process. Those have to be studied to arrive at a useful notion of language-parametric superimposition.

In the present paper, we will only deal with dynamic semantics. It is desirable to couple the adaptation of dynamic and static semantics. In the ideal situation, type safety of programs that involve superimposition should hold by construction. We might actually want to amalgamate static and dynamic semantics to be able to formulate certain kinds of pointcuts for superimposition that deal with types.

From here on, we will represent semantics definitions in Haskell. This immediately allows us to run these definitions as interpreters.

3 An example

We will now illustrate how to make the semantics of a very simple expression language fit for superimposition, and how to make use of superimposition. The language comprehends expression forms for integer constants, variables, and binary arithmetic expressions. Superimposition will be used in a way to catch division by zero.

In Fig. 1, the semantics of the expression language is defined in the denotational style. We define a type of denotations for each syntactical category, namely *Dexpr* for *Expr*, and *Dbinop* for *Binop*. The type *Dexpr* expresses that the meaning of an expression is a mapping from environments to values. The environment maps variable identifiers (i.e., strings) to values. The type *Dbinop* expresses that the meaning of a binary operator is a function that maps two values to a single value. We define one meaning function for each syntactical category using case discrimination according to the syntactical forms. The definition is compositional, that is, the meaning of a construct is defined in terms of the meanings of its subconstructs only, but not the subconstructs themselves.

We will now add superimposition to the semantics according to the three steps listed in Sec. 2. The first step is to elaborate the semantic functions to carry advice. One way to realise this step is to perform 'monad introduction', that is, to migrate to monadic style. This step is independent of the fact that we deal with semantics and superimposition. In principle, any family of recursive function definitions can be turned into monadic style. In [7], we define such a transformation. So the denotation types in the types of the semantic functions have to be wrapped by a monad. In the semantic equations, all compound denotations are sequentialised, and they are recomposed by "≫=". Without loss of generality, we assume a call-by-value order. The result of this step is shown in Fig. 2. Notice the elaborated types of the semantic functions, which involve a monad m. Also notice the monadic *do*-sequence for the compound meaning of a binary expression, and several occurrences of *return*. Instantiating the monad m by the identity monad, and β-reducing away sequentiality, we would obtain the original interpreter.

```
Expression syntax
  data Expr   =  Const Int | Var String | Bin BinOp Expr Expr
  data BinOp  =  Div | ···

Denotations
  type Dexpr  =  Env → Val
  type Dbinop =  Val → Val → Val

Variable environments
  type Env    =  String → Val

Values incl. an error value
  type Val    =  Maybe Int

Expression evaluation
  expr :: Expr → Dexpr
  expr (Const int) ρ    =  Just int
  expr (Var id) ρ       =  ρ id
  expr (Bin o e1 e2) ρ  =  binop o (expr e1 ρ) (expr e2 ρ)

Interpretation of binary operators
  binop :: BinOp → Dbinop
  binop Div (Just x) (Just y)  =  Just (x ‘div‘ y)
  binop ...                    =  ...
  binop _ _ _                  =  Nothing
```

Figure 1. Denotational semantics of a simple expression language. Because it is a Haskell program it can be viewed as an interpreter as well.

```
Evaluation function
  expr               ::  Monad m ⇒ Expr → m Dexpr
  expr (Const int)   =   return (λρ → Just int)
  expr (Var id)      =   return (λρ → ρ id)
  expr (Bin o e1 e2) =   do de1 ← expr e1
                            de2 ← expr e2
                            dop ← binop o
                            return (λρ → dop (de1 ρ) (de2 ρ))

Interpretation of binary operators
  binop        ::  Monad m ⇒ BinOp → m Dbinop
  binop Div    =   return (λv1 v2 →
                       case (v1, v2) of
                            (Just x, Just y) → Just (x ‘div‘ y)
                            _ → Nothing)
  binop ...    =   ...
```

Figure 2. The interpreter in monadic style

Expression evaluation; adapted

$$\begin{array}{lcl}
\mathit{expr}, \overline{\mathit{expr}} & :: & (\mathit{Superimposable}\ m\ \mathit{Expr}\ \mathit{Dexpr}, \\
 & & \ \mathit{Superimposable}\ m\ \mathit{BinOp}\ \mathit{Dbinop}) \\
 & \Rightarrow & \mathit{Expr} \rightarrow m\ \mathit{Dexpr} \\
\mathit{expr}\ (\mathit{Const}\ \mathit{int}) & = & \mathit{return}\ (\lambda\rho \rightarrow \mathit{Just}\ \mathit{int}) \\
\mathit{expr}\ (\mathit{Var}\ \mathit{id}) & = & \mathit{return}\ (\lambda\rho \rightarrow \rho\ \mathit{id}) \\
\mathit{expr}\ (\mathit{Bin}\ o\ \mathit{e1}\ \mathit{e2}) & = & \mathbf{do}\ \mathit{de1} \leftarrow \overline{\mathit{expr}}\ \mathit{e1} \\
 & & \quad \mathit{de2} \leftarrow \overline{\mathit{expr}}\ \mathit{e2} \\
 & & \quad \mathit{dop} \leftarrow \overline{\mathit{binop}}\ o \\
 & & \quad \mathit{return}\ (\lambda\rho \rightarrow \mathit{dop}\ (\mathit{de1}\ \rho)\ (\mathit{de2}\ \rho)) \\
\overline{\mathit{expr}}\ e & = & e\ \mathbf{:\text{-}}\ \mathit{expr}\ e
\end{array}$$

Interpretation of binary operators; adapted

$$\begin{array}{lcl}
\mathit{binop}, \overline{\mathit{binop}} & :: & \mathit{Superimposable}\ m\ \mathit{BinOp}\ \mathit{Dbinop} \\
 & \Rightarrow & \mathit{BinOp} \rightarrow m\ \mathit{Dbinop} \\
\mathit{binop}\ \mathit{Div} & = & \mathit{return}\ (\lambda v1\ v2 \rightarrow \\
 & & \qquad \mathbf{case}\ (v1, v2)\ \mathbf{of} \\
 & & \qquad\quad (\mathit{Just}\ x, \mathit{Just}\ y) \rightarrow \mathit{Just}\ (x\ \text{‘}\mathit{div}\text{‘}\ y) \\
 & & \qquad\quad \_ \rightarrow \mathit{Nothing}) \\
\mathit{binop}\ \ldots & = & \ldots \\
\overline{\mathit{binop}}\ o & = & o\ \mathbf{:\text{-}}\ \mathit{binop}\ o
\end{array}$$

Figure 3. The interpreter with reflective denotations

The second step in our procedure for adding superimposition to a semantics is to turn the semantic equations into reflective style by invoking the " **:-** " operator prior to case discrimination. For each semantic function, we define an overlined version that adds the application of " **:-** ", e.g., $\overline{\mathit{expr}}$ complements *expr*. (Alternatively, we could adapt *all* existing equations to invoke " **:-** " as described in Sec. 2.) In the semantic equations, all references to the original semantic functions are replaced by references to the overlined ones. The result of this step is shown in Fig. 3. Notice the new definitions of $\overline{\mathit{expr}}$ and $\overline{\mathit{binop}}$. Also notice the references to $\overline{\mathit{expr}}$ and $\overline{\mathit{binop}}$ in the semantic equations for *expr*.

Recall that the operator " **:-** " models transformation of reflective denotations. Since a language normally comprehends several syntactical domains and corresponding denotation types, the operator " **:-** " needs to be overloaded for all couples of syntactical domains and associated denotation types. So a registry is actually a *tuple* of denotation transformers — one for each denotation type. For a given denotation type, the operator is meant to look up the corresponding denotation transformer from the registry tuple and to apply it to the term and the denotation at hand. This can be conveniently represented in Haskell using a class for overloading. So we place the operator " **:-** " in a class *Superimposable*, which subclasses the standard class *Monad* as follows:

$$\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{Superimposable}\ m\ s\ d \\
\quad \mathbf{where} \\
\qquad (\mathbf{:\text{-}}) :: s \rightarrow m\ d \rightarrow m\ d
\end{array}$$

There are three parameters: *m* is the type constructor of the monad for the registry, *s* is a syntactical domain, *d* is the type of denotations for *s*. We will see in a second that the instances of the *Superimposable* class follow a simple scheme.

The third step in our procedure for adding superimposition to a semantics is to integrate a construct for binding advice. This includes the obligation to opt for a specific instance of a *Superimposable* monad. We will now provide a general realisation of the third step including its illustration for the simple expression language. As always with our method, the language designer might bypass the language-parametric approach if a more language-specific form of superimposition is favoured.

The registry domain constructor

$$\mathbf{type}\ R = \mathit{Reader}\ \mathit{Hook}$$

Types of hooks for superimposition

$$\begin{array}{l}
\mathbf{data}\ \mathit{Hook} = \mathit{Hook}\ (\mathit{Hexpr}, \mathit{Hbinop}) \\
\mathbf{type}\ \mathit{Hexpr} = \mathit{Expr} \rightarrow R\ \mathit{Dexpr} \rightarrow R\ \mathit{Dexpr} \\
\mathbf{type}\ \mathit{Hbinop} = \mathit{BinOp} \rightarrow R\ \mathit{Dbinop} \rightarrow R\ \mathit{Dbinop}
\end{array}$$

The identity Hook

$$\mathit{idHook} = \mathit{Hook}\ (\lambda e\ ce \rightarrow ce, \lambda o\ co \rightarrow co)$$

Run a reflective denotation

$$\begin{array}{l}
\mathit{run} :: R\ d \rightarrow d \\
\mathit{run}\ d = \mathit{runReader}\ d\ \mathit{idHook}
\end{array}$$

Instances of *Superimposable* class

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Superimposable}\ (\mathit{Reader}\ \mathit{Hook})\ \mathit{Expr}\ \mathit{Dexpr}\ \mathbf{where} \\
\quad t\ \mathbf{:\text{-}}\ d = \mathit{ask} \gg\!= \lambda(\mathit{Hook}\ (h, \_)) \rightarrow h\ t\ d \\
\mathbf{instance}\ \mathit{Superimposable}\ (\mathit{Reader}\ \mathit{Hook})\ \mathit{BinOp}\ \mathit{Dbinop}\ \mathbf{where} \\
\quad t\ \mathbf{:\text{-}}\ d = \mathit{ask} \gg\!= \lambda(\mathit{Hook}\ (\_, h)) \rightarrow h\ t\ d
\end{array}$$

Figure 4. Uniform definition of registry

Syntax of the superimposition construct

$$\mathbf{data}\ \mathit{Expr} = \ldots \mid \mathit{Adapts}\ \mathit{Hook}\ \mathit{Expr}$$

Semantics of the superimposition construct

$$\overline{\mathit{expr}}\ (\mathit{Adapts}\ h\ e) = \mathit{local}\ (\mathit{const}\ h)\ (\overline{\mathit{expr}}\ e)$$

Figure 5. Uniform integration of a superimposition construct

In Fig. 4, we define a specific monad *R* that models a registry for superimposition in our example. In fact, we choose the environment monad (aka *Reader* in Haskell).[1] The type *Hook* is a product with two components, one for each denotation type. Each such hook component is a denotation transformer. The received denotation is the normal denotation, whereas the computed denotation is the revised, ultimate denotation. The types make clear that a denotation transformer also receives a syntactical entity, which can contribute to the decision whether to replace or to preserve the normal denotation. In the figure, we also define an identity hook (i.e., *idHook*), which models that the normal denotation is preserved regardless of the 'join point' (i.e., the syntactical form at hand). Running a reflective denotation is like 'running' the *Reader* monad with *idHook* as initial registry; see *run*. The last few lines in Fig. 4 instantiate the *Superimposable* class for our example semantics. That is, " **:-** " is defined by looking up the denotation transformers from the environment and by applying the relevant transformer to the ingredients of the given reflective denotation.

In Fig. 5, we complete the extension of the sample semantics by adding a specific case to the semantic function for expressions. This new equation provides the most simple and uniform kind of a superimposition construct. The meaning of *Adapts h e* is that the hook *h* adapts the denotation for the expression *e* and all denotations for

[1] We recall the operations of the *Reader* monad:

ask	::	$m\ r$	-- read environment
local	::	$(r \rightarrow r) \rightarrow m\ x \rightarrow m\ x$	-- locally adapt environment

```
noDivByZero = Hook (λe d → d, noDivByZero′)
  where
    noDivByZero′ :: BinOp → R Dbinop → R Dbinop
    noDivByZero′ Div d =
      do d′ ← d
         return (λv1 v2 → case (v1, v2) of
                           (_, Just 0) → Nothing
                           _ → d′ v1 v2
                )
```

Figure 6. A hook for superimposition to catch division-by-zero

subconstructs of *e*. The use of the *local* operator makes it as clear as crystal that the hook *h* is only used for the interpretation of *e*. The use of *const* makes clear that previous bindings will be replaced by the new hook. We will investigate alternatives in the next section.

In Fig. 6, we define a hook for catching division-by-zero for any interpretation of *Div*. To this end, the second argument of the denotation is checked to be "0", and if this is the case, then the error value *Nothing* is returned. Otherwise, the original denotation is retained. So finally, we can demonstrate superimposition in action. To this end, let us consider the following program together with an environment for the used variables:

```
myexp = Bin Div (Const 42) (Var "myvar")
myenv = λid → if id ≡ "myvar" then Just 0 else Nothing
```

Using the original denotational semantics as an interpreter for this program, we will obviously encounter a division-by-zero run-time error. Using the aspect-oriented interpreter, we can catch division by zero. The following program execution returns *Nothing*:

$$run\ (\overline{expr}\ (Adapts\ noDivByZero\ myexp))\ myenv$$

4 Design space exploration

We will now walk through a few locations in the design space for a language semantics with superimposition. This will further substantiate the generality of our method, and it will clarify how it can be customised for a specific language at hand.

Binding policies We will first discuss different binding policies for advice. The policy that we have seen above employs an environment to carry advice. Here the affected program fragment is explicitly part of the binding construct. Also, we favoured the replacement of previous bindings by the new binding. Both design decisions can be altered. We will first discuss cumulative advice binding as opposed to replacement semantics before. We will then discuss the use of a state for the registry as opposed to an environment before.

In Fig. 7, we provide a new definition of the *Adapts* construct; see Fig. 5 for the original definition. We *chain* the previous binding and the new binding (cf. "∘"). The new binding gets into control but if it wanted to resort to the standard denotation, it actually accesses the denotation as processed by the previous binding. It is now not too difficult to think of further binding policies. For example, we could favour denotation transformers with yet another denotation argument for the standard denotation prior to any adaptation by previous bindings. This way, newly installed hooks could abandon previously installed hooks.

```
Syntax of the superimposition construct
  data Expr = ... | Adapts Hook Expr

Semantics of the superimposition construct
  expr‾ (Adapts (Hook (he, ho)) e) = local chain (expr‾ e)
    where
      chain (Hook (he′, ho′)) = Hook (λe → he e ∘ he′ e,
                                      λo → ho o ∘ ho′ o)
```

Figure 7. Superimposition with cumulative advice binding

```
Registry-aware computations
  type R = State Hook

Values incl. Void for pure side effects
  data Val = ... | Void

Syntax of the superimposition construct
  data Expr = ... | HookUp Hook

Semantics of the superimposition construct
  expr‾ (HookUp h) = put h >>= λ() → return (return Void)
```

Figure 8. Superimposition with a state for advice binding

In Fig. 8, we use the *State* monad as opposed to the *Reader* monad for the registry; see Fig. 4 for the original definition.[2] Here we assume that the language semantics at hand provides a notion of a purely side-effective computation. Hence, there is a designated result value *Void*. The construct for superimposition now also takes a different form because we do not list the affected program fragment, but we simply register advice along the execution of the program. So the construct for binding advice is of the form *HookUp h* with the intended semantics that the hook *h* as installed as registry at the time when the *HookUp* expression is executed. As one can see, the expression evaluates to *Void*. A problem with this approach is that the base semantics and the *do*-sequences for the introduced registry monad might accidentally disagree on the order of computation. We will come back to this problem in a minute.

Effect composition So we have seen that both an environment and a state make sense for carrying advice. Capturing this variation point in a monad parameter is a good idea because the superimposition level might even deal with further effects than just carried advice. For example, we might want to maintain dynamic join point information [13], or we might want to reflect on the success and failure of denotation transformation. Regardless of the choice monadic-style semantics vs. hard-wired effects, a discussion of the relationship between the superimposition level and the base semantics is in place.

By default, we assume that the semantics is made fit for carrying advice without looking at the denotation types. For example, even if the original semantics is already in monadic style, we can perform monad introduction. This will result in nested monadic style. In Fig. 9, this is illustrated for a variation on our expression language. The variation provides an *Assign* statement the semantics of which relies on the *State* monad for the variables in a program. The reflective denotation for an *Assign* expression is a nested monadic computation. At the top level, the computations for enabling super-

[2]We recall the operations of the *State* monad:

```
get  ::  m s        -- read state
set  ::  s → m ()   -- write state
```

```
Syntax extension for assignments
  data Expr  =  ... | Assign String Expr

Revision of expression denotations
  type Dexpr  =  State Env Val

Semantics of assignments
  expr (Assign id e') = do de' ← $\overline{expr}$ e'
                           return (do ρ ← get
                                      v ← de'
                                      put (λid' → if id ≡ id'
                                                    then v
                                                    else ρ id')
                                      return v)
```

Figure 9. The *Superimposable* monad on top of a base monad

imposition are arranged in a *do*-sequence. The inner *do*-sequence directly models the semantics of assignment. That is, the state is looked up with *get*, the right-hand side of the assignment is evaluated to *v*, the state is updated in the point for the variable *id*, the updated state is '*put* back', and the right-hand side value *v* is *return*ed as the value of the assignment.

These nested *do*-sequences pinpoint a problem. Suppose, we use a state for the registry; recall Fig. 8. A subexpression e_1 might hook up another subexpression e_2 while e_2 would be normally executed before e_1. That is, the nested *do*-sequences could disagree on the order of computation. Note that the inner sequence represents the base semantics whereas the outer sequence was established by systematic monad introduction. To enforce a common order, we should transform the monad in the base semantics to integrate the registry or any other superimposition effect as well. We could even elaborate an existing effect in the base semantics, e.g., an environment or a state, so that it accomplishes the registry as well. If the base semantics is not in monadic style, then it is not really prepared for such an amalgamation of effect spaces. In the view of these problems, our earlier choice of an environment monad for the registry is more favourable. The ordering problem is here a non-issue because advice binding is local with respect to a given term.

Intercepting invocations Our approach allows us to intercept *any* point of the program execution in the sense of syntactical fragments. It is at the heart of AOP to intercept invocations of methods or other procedural abstractions. So we want to briefly examine how this looks like in our setting. In Fig. 10, we further extend our expression language to accomplish a form of named function application. The semantic equation for function application is already prepared to carry advice. (We again use nested monadic style.) We use a helper function *apply* to apply a function-type value to an argument. For brevity, we do not define any expression form for function abstraction (i.e., λ-abstraction), but we assume that the environment can hold functions, e.g., a function "*div*". At the bottom of Fig. 10, we define a hook *noDivByZero*, which intercepts applications of the "*div*" function to catch division by zero. Notice the plain use of pattern matching for filtering out the relevant (nested) function application. The constructed denotation returns *Error* if the second argument of "*div*" is "0", and otherwise it applies the original binding of "*div*". This hook looks a bit verbose because it reconstructs the normal denotation to a large extent. This could be captured by a reusable operator for 'function-application interception'.

```
Syntax extension for function applications
  data Expr =  ... | Apply Expr Expr

Functions evaluate to functions
  data Val   =  ... | Fun (Val → Val)

Adapted semantics of function application
  expr (Apply e1 e2) = do de1 ← $\overline{expr}$ e1
                          de2 ← $\overline{expr}$ e2
                          return (do v1 ← de1
                                     v2 ← de2
                                     return (apply v1 v2)
                                 )

Helper for function application
  apply :: Val → Val → Val
  apply (Fun f) val = f val
  apply _ _ = Error

Another division-by-zero catcher
  noDivByZero :: Expr → R Dexpr → R Dexpr
  noDivByZero (Apply (Apply (Var "div") e1) e2) d =
   do de1 ← $\overline{expr}$ e1
      de2 ← $\overline{expr}$ e2
      return (do ρ ← get
                 v1 ← de1
                 v2 ← de2
                 case v2 of
                  Int 0 → return Error
                  _ → return (apply (apply (ρ "div") v1) v2)
             )
  noDivByZero _ d = d
```

Figure 10. Intercepting a function application

5 Related work

For distributed systems (of communicating processes), there exists a notion of superimposition [3, 6, 10] which is (like aspects in AOP) orthogonal to the usual breakdown of modules. This sort of superimposition contributes to the theoretical basis of AOP. Another abstract, formal model of AOP is provided in [4]. It is based on execution monitors for the events that correspond to the points of interest along the program execution. Another formal semantics of AOP is based on CSP with CSP synchronisation sets as join points [1]. All this work differs from ours in that we start from an ordinary denotational semantics, and make it fit for superimposition in a systematic manner. That is, we do not resort to any designated formal model, but we just stay in the denotational setting.

Our approach to reflect on the syntactical patterns along program interpretation is inspired by the event grammars in [2]. Auguston suggests to formalise the execution of a program in a language in terms of an event grammar. Such a behavioural model can then be used to superimpose functionality on the event traces of a program, e.g., to check assertions, or to perform debugging. This approach has been used in the development of several debugging tools. Our notion of 'reflective denotations' is a semantic transposition and a strong generalisation of a tweaked monadic-style of functional programming proposed by Meuter in [9]. In this style, the programmer informs a non-standard monadic bind combinator about the names of functions that are applied to intermediate results. These names can be viewed as (explicit) join points. By contrast, we prepare a semantics in a way that a 'superimposable' monad can revise denotations for syntactical patterns. Our method is based on a systematic transformation as opposed to an encoding style.

There is an enormous amount of related work on reflection [11, 12, 5]; its relevance for AOP is generally acknowledged. We have not seen a generic method to systematically elaborate a denotational semantics for AOP-like reflection in the available literature. The reflection literature is normally concerned with some kind of staged interpretation as opposed to the provision of a superimposition construct. However, it seems that our approach could take great advantage of the reflective theory for the purpose of formal reasoning on aspect-oriented programs. Also, ideas on 'full computational reflection' are of use to further generalise our approach.

6 Concluding remarks

The described method defines how to extend an ordinary language *semantics* so that one obtains an aspect-oriented language semantics. We call this achievement '*superimposition for free*'. Technically, it is based on a 'reflective denotation style'. Accidentally, the approach also suggests a normative style of *aspect-oriented functional programming*, but this has to be discussed elsewhere. The aspect-oriented programming terms are instantiated for 'superimposition for free' languages as follows:

Static join point	=	Syntactical pattern
Dynamic join point	=	Computation on syntactical pattern
Point cut	=	Pattern matching predicate
Advice	=	Denotation transformer
Program execution	=	Monadic do-sequence
Aspect	=	Hook for superimposition
Dynamic weaving	=	Registry update
Static weaving	=	Partial evaluation

We contend that this is a rather simple, uniform, and general way to define aspect-oriented language semantics. We are also willing to say that our approach can be seen as another definition of reflection.

The to-do list for an exhaustive treatment of the subject is long:

- Transpose the method to static semantics.
- Cover the standard forms of dynamic join points.
- Make even the denotation transformers reflective.
- Recover compositionality in some way.
- Cover SOS in addition to denotational semantics.
- ...

Acknowledgement

I am very grateful to the three anonymous FOAL 2003 workshop referees for their encouraging and constructive remarks. I am also grateful to the participants of the Belgian-Dutch poster workshop on AOSD, on 21 January 2003 in Twente, with whom I had stimulating discussions on the subject of the paper.

7 References

[1] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Berlin, Heidelberg, and New York, Sept. 2001. Springer-Verlag.

[2] M. Auguston. Program behavior model based on event grammar and its application for debugging automation. In M. Ducassé, editor, *AADEBUG, 2nd International Workshop on Automated and Algorithmic Debugging*, pages 277–291, Saint Malo, France, 22–24 May 1995. IRISA-CNRS.

[3] L. Bougé and N. Francez. A compositional approach to superimposition. In ACM, editor, *Proc. of the 1988 conference on Principles of programming languages (POPL'88), January 13–15, 1988, San Diego, CA*, pages 240–249, New York, NY, USA, 1988. ACM Press.

[4] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*, pages 170–186. Springer-Verlag, Sept. 2001.

[5] S. Jefferson and D. P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181–202, May/June 1996.

[6] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, Apr. 1993.

[7] R. Lämmel. Reuse by Program Transformation. In G. Michaelson and P. Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

[8] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[9] W. D. Meuter. Monads as a theoretical foundation for AOP. In S. Mitchell and J. Bosch, editors, *Workshop Reader, ECOOP'97*, volume 1357 of *LNCS*. Springer-Verlag, 1998.

[10] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 28–40. ACM Press, 2002.

[11] B. C. Smith. Reflection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, ACM, Jan. 1984.

[12] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In R. P. Gabriel, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 298–307, Cambridge, MA, Aug. 1986. ACM Press.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Report 02-06, Dept. of Comp. Sc., Iowa State Univ., pages 1–8, Apr. 2002.