Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3592-3607
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © J.UCS

Disentangling Denotational Semantics Definitions

Fabio Tirelo
(Instituto de Informática, Pontifícia Universidade Católica de Minas Gerais
Brazil
ftirelo@pucminas.br)

Roberto S. Bigonha
(Departamento de Ciência da Computação, Universidade Federal de Minas
Gerais, Brazil
bigonha@dcc.ufmg.br)

João Saraiva
(Departamento de Informática, Universidade do Minho, Portugal
jas@di.uminho.pt)

Abstract: Denotational semantics is a powerful technique to formally define programming languages. However, language constructs are not always orthogonal, so many semantic equations in a definition may have to be aware of unrelated constructs semantics. Current approaches for modularity in this formalism do not address this problem, providing, for this reason, tangled semantic definitions. This paper proposes an incremental approach for denotational semantic specifications, in which each step can either add new features or adapt existing equations, by means of a formal language based on function transformation and aspect weaving.

Key Words: semantics of programming languages, denotational semantics, modularity, aspect-oriented definitions
Category: D.3.1, D.3.3, F.3.2, F.3.3

1 Introduction

Formal semantics of large scale programming languages is inherently complex due to the large number of crosscutting details that must be coped with. It is then desirable that such specifications be modular and extensible, and be written in an incremental way, so that constructs may be successively added to the definition of a core language. Moreover, this incremental process must not force the redefinition of previously written modules, and additionally must require that the language designer use only simple features and techniques.

However, large scale programming languages usually are composed by constructs which are not always orthogonal, and therefore providing separate definitions for them is not trivial. An example of this problem is the definition of method calls and exception handling in Java: not only must the definition of the *return* statement specify its expected behavior, but also it has to be aware of possible *finally* blocks which must be executed before restoring the execution control to the caller function, either by normal return or via exception handling mechanism. Furthermore, because the semantics of a program is produced by the semantic equations in a top-down way, each construct may be responsible for preparing context for each possible constituent.

As a consequence, not only are the semantic equations responsible for specifying the meaning of the constructs, but also they must define how such constructs interact with each other. Moreover, for defining an interaction between two constructs, at least one of them must be aware of the existence of the other. For instance, the problem of *return* statements inside *try* blocks may be solved by redefining the continuation associated with the sequencer so that the *finally* block is executed before returning. Since in traditional approaches of denotational semantic specifications (see Section 2) there is a one-to-one mapping of language constructs to semantic equations, such interactions definitely induce tangled elements in at least one semantic equation[1].

This property directly impacts the modularity of the language's denotational semantics definition, because the description of one construct must contain elements of unrelated ones, which violates the principle of module high cohesion. In addition, the incremental definition of a language usually requires that previously defined modules be rewritten whenever a new construct is defined, so that the whole writing and rewriting process becomes tedious and error-prone. From the reader point of view, crucial information about the language may be obscured by several details in the semantic equations, which makes it hard to fully understand some constructs.

The main contribution of this paper is an aspect-oriented-based technique for improving the process of incrementally defining programming language denotational semantics. In this approach, constructs are defined in a two phases: first, the construct is separately defined, and the semantic equations may not be aware of other constructs; then, the influence of other constructs on it is specified. Understanding such specifications is also a two-phases process. First the reader can understand the key concepts on the language constructs; in this first reading the relation among those constructs is abstract. After having acquired expertise on the individual constructs, the reader can focus on how such constructs interact, to get the whole picture of the definition.

2 Current Approaches

One of the first steps to the modularity of denotational semantics has been made by Mosses in the Action Semantics [Mosses 1977]. Further attempts to

[1] It is a direct consequence of the pigeon-hole principle. However, this is not true for systems based on structural operational semantics, because more than one clause may be used to separately define behavior of a construct.

improve the modularity of denotational semantics have been made since then, with highlight to Monadic Semantics [Liang et al. 1995] and Monadic Action Semantics [Wansbrough and Hamer 1997]. Recently problems related to separation of concerns in semantic specifications were addressed by De Moor et al. [Moor et al. 2000] and Mosses [Mosses 2004, Mosses 2005]. This work improves the results of those contributions by presenting a modular mechanism for defining and transmitting context information among programming languages constructs.

The existence of a one-to-one mapping between language constructs and semantic equations, which leads to the lack of modularity as discussed in Section 1, is found in Action Semantics, Monadic Semantics, and Monadic Action Semantics. In fact, both action notation and monads provide elegant mechanisms to abstract the structure of context information for antecedents and destinations. However, structural context information is usually propagated by means of stores and environments, so that such propagation appears tangled in the semantic equations.

De Moor et al. [Moor et al. 2000] proposes an aspect-oriented based technique to improve the modularity of attribute grammar, which can also be applied to semantic definitions. In that model, attributes may be defined in separate sections and *"can be woven together to form a pure attribute grammar"*. By letting attributes be defined with aspect support, it is possible, for instance, to create a definition for repetition which is vague with respect to sequencers. However, interactions among language constructs may need information not available for attribute definition, specially if they are decoupled from the syntactic structure. For example, in the following piece of Java code, the *throw* in function f to the *catch* in function g cannot be expressed by neither *inherit*, *synthesize*, nor *chain* clauses, because there is no syntactic relation between the constructs.

```
void f() { throw new E(); }
void g() { try { f(); } catch (E e) { ... } finally { ... } }
```

Mosses [Mosses 2004] provides a model inspired on monadic semantics for defining modular structural operational semantics (MSOS) of programming languages. As new constructs are added to a specification, the context may evolve without requiring that previously written equations be redefined. Context information is transmitted as labels of transition rules, and modularity and extensibility are derived by letting them abstract from the structure of labels. The result is a set of abstract transition rules, without explicit context information to get in the way. Moreover, as SOS programmers to define separately the interaction among constructs, it is also possible in MSOS. However, some interactions may need to be defined by listing all possible cases, as in the definition of Java provided by [Cenciarelli et al. 1999]; even without defining repetition and sequencers, there are 33 transition rules to define function call and return, exception handling, and their interactions[2].

Reusability can also be improved by means of the constructive approach proposed by Mosses [Mosses 2005]. In this approach, for a given language, a representative set of abstract constructs is formally defined, and concrete constructs may be translated into the defined abstractions, so that their semantics are straightforwardly obtained. Furthermore, sequencers can be defined as exceptions to be handled by an exception handling mechanism associated with the *while* statement. However, by doing it, elements for sequencers handling remain interleaved in the definition of the *while* statement, and therefore bringing into the initial definition of the concrete statement concerns about the effects of such exceptions.

3 Incremental Definitions

Programming languages semantic definitions are large and complex systems which are better defined in an incremental way, so that new constructs and behaviors are defined upon existing ones. For instance, when teaching a programming language, it is worthwhile to abstract away advanced concepts to explain basic constructs first; later the introduction of such concepts may redefine previously explained elements, while preserving their basic nature. Such *vague* explanation is desirable because it usually requires less effort to learn the language concepts.

Let a semantic specification be the quaduple $S = (G, D, \tau, \rho)$, where G is the language abstract grammar, D is the set of semantic domains, τ is a type environment which maps semantic function names into their domains, and ρ is the environment which maps semantic function names into their definitions, i.e, the semantic equations. To achieve abstractness, environment ρ maps each function name into a set of defining clauses, each one represented by its list of patterns and its expression.

Function applications occurring inside function bodies are indirectly performed in an environment ρ, which dynamically binds function identifiers to their definitions. Thus, if specification S is composed by functions $f_1, f_2, \cdots, f_n$, where each f_i is defined by at least one clause with the form $f_i\ p_{i1} \cdots p_{ik} = e_i$, then the corresponding clause in the environment is[3] $([\rho, p_{i1}, \cdots, p_{ik}], e_i[(\rho\ f_j)\ \rho/f_j, \forall j])$. If function f is defined by means of cases based on pattern matching, then the environment argument is included in each case. In addition, the list of patterns

[2] In fact, those rules only define the behavior of a return statement inside a *try* block, without specifying its behavior inside a *catch* block; then, by those rules, if a *return* is executed inside a *catch* block the corresponding *finally* block is not executed.

[3] The elements of this clause represents nodes of a specification's abstract syntax tree. Notation $e[e'/x]$ represents expression e in which all free occurrences of x are replaced by e'. In expression $\rho\ f$, consider f be the function identifier and not the actual function itself.

is considered as if arguments were not ruled out by η-reductions, and *don't care* patterns (like Haskell's _) were replaced by fresh identifiers.

For instance, if a specification is composed solely by function $\mathcal{E} : \mathit{Exp} \rightarrow \mathit{Env} \rightarrow \mathit{Val}$, such that $\mathcal{E}[\![\mathit{Id}]\!]\ r = r\ \mathit{Id}$, and $\mathcal{E}[\![E_1 + E_2]\!]\ r = \mathcal{E}[\![E_1]\!]\ r + \mathcal{E}[\![E_2]\!]\ r$, then type environment τ is defined as $\tau = \{\mathcal{E} \mapsto (\mathit{Exp} \rightarrow \mathit{Env} \rightarrow \mathit{Val})\}$, and definition environment ρ is defined as $\rho = \{\mathcal{E} \mapsto \{\mathit{clause}_1, \mathit{clause}_2\}\}$, where $\mathit{clause}_1 = ([\rho, [\![\mathit{Id}]\!], r], r\ \mathit{Id})$ and $\mathit{clause}_2 = ([\rho, [\![E_1 + E_2]\!], r], ((\rho\ \mathcal{E})\ \rho)\ [\![E_1]\!]\ r + ((\rho\ \mathcal{E})\ \rho)\ [\![E_2]\!]\ r)$.

An incremental definition of the semantics of a language is defined as a sequence $S_0, S_1, \cdots, S_n$, where each S_i is a semantic specification of a language's subset, and S_{i+1} includes further behavior to S_i. Each new specification may add new elements, but sometimes it may be necessary to adapt previous existing equations. Given a specification S_i, a new specification $S_{i+1} = t(S_i) + \Delta S_i$ is obtained from S_i by applying to it a transformation function t and including the elements defined in ΔS_i.

A *transformation* is a function t that maps semantic specifications into semantic specifications. This function is meant to adapt the behavior of semantic equations. The effect of such function is to define new environments τ', ρ' mapping each function to its new definition, so that $t\ (G, D, \tau, \rho) = (G, D, \tau', \rho')$. An *inclusion* into a specification S, denoted by ΔS, represents new elements to be added to the specification, usually elements concerning with new language constructs.

The working example used throughout this paper consists of a specification S_i of a language L composed by expressions, declarations, and commands, whose abstract syntax, and semantic functions and equations for the relevant constructs to the discussion are presented in Figure 1. In these equations[4], the definition of the *while* statement is vague with respect to the existence of sequencers. If specification S_{i+1} defines sequencer *break*, it is necessary to adapt the *while* equation to prepare the context in which the sequencer is executed. The corresponding inclusions consist on defining a new grammar rule for the *break* sequencer and a new semantic equation to define it.

This paper concentrates on function transformations, which is its main contribution. Other kinds of inclusions can be easily achieved by using ordinary modularity features of programming languages and, therefore, are not further presented. Although the running example is based on traditional continuation semantics, the proposed technique is suitable for using with other modularity improving techniques, such as monads, as shown in the case study of Section 8.

A definition increment may affect existing semantic functions and equations by requiring: (i) function signatures be redefined to include new arguments, to change the type of some function argument, or to change return types; (ii) func-

[4] FIX represents the fix point operator.

Abstract syntax:	Semantic Functions:
$P \in Prog \rightarrow D; C$	$\mathcal{P} : Prog \rightarrow Ans$
$C \in Com \rightarrow \textbf{while}\ E\ C \mid C_1; C_2 \mid$	$\mathcal{C} : Com \rightarrow Env \rightarrow Cc \rightarrow Cc$
$E \in Exp \rightarrow \cdots$	$\mathcal{E} : Exp \rightarrow Env \rightarrow (Val \rightarrow Cc) \rightarrow Cc$
$D \in Dec \rightarrow \cdots$	$\mathcal{D} : Dec \rightarrow Env \rightarrow (Env \rightarrow Cc) \rightarrow Cc$

Semantic Equations:

$\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.stop))\ s_0$
$\mathcal{C}[\![C_1; C_2]\!]\ r\ c = \mathcal{C}[\![C_1]\!]\ r;\ \mathcal{C}[\![C_2]\!]\ r\ c$
$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r = \textsc{fix}\ \lambda fc.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ (f\ c)\ \textbf{else}\ c$
Other equations defining $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{D}$

Figure 1: Working Example of the Paper – Only Relevant Constructs for the Discussion Are Presented.

tion arguments or its return values be decorated[5] in order to handle unpredicted situations or to conform the equations to signature changes; (iii) some equations be completely redefined, when no automatic transformation is implied. A function is *promoted* with respect to a transformation t if it is subject to any modification defined in t. Function transformations may be defined by means of the following construction, whose constituents are defined in Sections 4-7:

transformation *transformation-name*
- **signature** $f_1 : T_1$ **to** $T'_1, f_2 : T_2$ **to** $T'_2, \cdots, f_m : T_m$ **to** T'_m
- **default** $l_1 = c_1, l_2 = c_2, \cdots, l_n = c_n$
- **application** $l_1 \Rightarrow e'_1, l_2 \Rightarrow e'_2, \cdots, l_n \Rightarrow e'_n$
- **use** $l_1 \Rightarrow e_1, l_2 \Rightarrow e_2, \cdots, l_n \Rightarrow e_n$
- **replace** $f_1\ p_{11}\ \cdots\ p_{1k_1}$ **by** $e_1, \cdots, f_n\ p_{n1}\ \cdots\ p_{nk_n}$ **by** e_n
- **redefine** $f_1\ p_{11}\ \cdots\ p_{1k_1}\ =\ e_1, \cdots, f_n\ p_{n1}\ \cdots\ p_{nk_n}\ =\ e_n$

Given a specification S and a transformation function t, $S' = t(S)$, is defined in two steps: the first step collects the transformations to be performed on each individual function and produces a sequence $\langle t_1, t_2, \cdots, t_m \rangle$, where each t_i can be an argument inclusion, a type redefinition, a decoration, or an equation redefinition; the second step creates the new environments by applying the transformations on each function. In such sequence, argument inclusions and type redefinitions are performed before decorations, which, on their turn, take place before equation redefinitions.

[5] Decoration has the meaning established in the GoF Design Patterns [Gamma et al. 1995]. In fact, it can be understood as the implementation of this pattern using *around advices* from Aspect-Oriented Programming [Kiczales et al. 1997].

4 Argument Inclusion

In the signature changing clauses of Section 3, each function f_i having type T_i must be transformed into a function of type T_i'. Labels may be assigned to constituents of each T_i, T_i' in order to denote inclusion of new arguments, or transformation of arguments into new ones. A labelled type has the form $(l : t)$, where l is a label, and t is a type expression.

A signature changing clause is valid when the following rules apply: (single occurrence) each label occurs at most once in each T_i, and at most once in each T_i'; (left consistency) if $(l : t_1)$ is a component of T_i, and $(l : t_2)$ is a component of T_j, then $t_1 = t_2$; (right consistency) if $(l : t_1')$ is a component of T_i', and $(l : t_2')$ is a component of T_j', then $t_1' = t_2'$; (left-right correspondence) if $(l : t)$ is a component of T_i, then $(l : t')$ is a component of T_i' for some type t'.

All new arguments included by the signature clauses are passed through recursive applications of the functions. In the absence of such value, a default value to be inserted in the original applications is defined by means of the default clause. In the default clauses, each l_i is a label for a type t_i in the right-hand side of a signature changing clause, and c_i is an expression of type t_i. For example, one possible solution to the problem of including the *break* sequencer is to include a new argument into semantic function $\mathcal{C}$ representing the continuation for a *break* statement, as defined by transformation *include_break_a*.

$$
\begin{array}{l}
\textbf{transformation } \mathit{include\_break\_a} \\
\quad \textbf{signature } \mathcal{C} : \quad \mathit{Com} \to \mathit{Env} \to \mathit{Cc} \to \mathit{Cc} \\
\qquad\qquad\qquad\qquad \textbf{to } \mathit{Com} \to \mathit{Env} \to (b : \mathit{Cc}) \to \mathit{Cc} \to \mathit{Cc} \\
\quad \textbf{default} \quad b = \lambda s.\mathit{error}
\end{array}
$$

To every application of function $\mathcal{C}$ a new third argument $\lambda s.\mathit{error}$ will be automatically inserted except when it is a recursive call, in which case the corresponding formal parameter is simply propagated. The application of transformation *include_break_a* produces the following modified versions of the semantic equations of Figure 1[6]:

$$
\begin{array}{l}
\mathcal{C} : \mathit{Com} \to \mathit{Env} \to \mathit{Cc} \to \mathit{Cc} \to \mathit{Cc} \\
\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.\mathit{error})\ (\lambda s.\mathit{stop}))\ s_0 \\
\mathcal{C}[\![C_1; C_2]\!]\ r\ b\ c = \mathcal{C}[\![C_1]\!]\ r\ b;\ \mathcal{C}[\![C_2]\!]\ r\ b\ c \\
\mathcal{C}[\![\textbf{while } E\ C]\!]\ r\ b\ c = (\textsc{fix}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if } v \textbf{ then } \mathcal{C}[\![C]\!]\ r\ b\ (f\ c')\ \textbf{else } c')\ c
\end{array}
$$

Sequencer break might then be defined by: $\mathcal{C}\ [\![\textbf{break}]\!]\ r\ b\ c = b$.

[6] Equations not affected by this transformation are not reproduced here. It is important to highlight that this version is still incomplete for the inclusion of the *break* sequencer requires the application of decorators.

4.1 Formal Aspects of Argument Inclusion

Transformations for argument inclusions are collected, and the following structures are defined: $\alpha = (\mathit{label}, \mathit{type}, \mathit{fmarks}, \mathit{def\text{-}value})$ is composed by the label of the new argument, its type, the list of all functions affected by the inclusion, and the default value of the argument; $\mathit{fmarks} = \mathit{function\text{-}name} \rightarrow (\mathsf{arg}, \mathit{type}^*, \mathit{type})$ maps function names into a pair consisting on a list of argument types and the result type of the function. The new argument is included after the list of argument types. When multiple argument inclusions are performed on a function, they are sequentially handled so that each inclusion considers the effect of the previous ones. For instance, the inclusion defined by the transformation *include_break_a* is represented by $\alpha_b = (b, Cc, \mathit{fmarks}, \lambda s.\mathit{error})$, where $\mathit{fmarks} = \{\mathcal{C} \mapsto (\mathsf{arg}, [Com, Env], Cc \rightarrow Cc)\}$.

Given an inclusion $\alpha = (x, t, \mathit{fmarks}, x_0)$ and specification environments τ and ρ, new environments τ' and ρ' are defined as[7]:

$$\begin{aligned}
&\tau' = \tau[t_1 \rightarrow \cdots \rightarrow t_k \rightarrow \underline{t} \rightarrow t'/f_i, \forall (f_i \mapsto (\mathsf{arg}, [t_1, \cdots, t_k], t')) \in \mathit{fmarks}] \\
&\rho' = \{(f_i \mapsto \text{MAP } (\mathit{include}\ \alpha)\ \mathit{clauses}) \mid \mathit{clauses} = \rho\ f_i, f_i \text{ bound in } \mathit{fmarks}\} \\
&\qquad \bigcup \big\{(f_i \mapsto \text{MAP } (\mathit{include}'\ \alpha)\ \mathit{clauses}) \mid \mathit{clauses} = \rho\ f_i, f_i \text{ unbound in } \mathit{fmarks}\big\}
\end{aligned}$$

Function *include* performs the inclusion of the argument in the functions being promoted and is defined as $\mathit{include}\,(x, \_, \mathit{fmarks}, \_)\ (\rho : \mathit{pats}, \mathit{exp}) = (\rho : \mathit{pats}', \mathit{exp}[\rho''/\rho])$, where pats' corresponds to the inclusion of identifier pattern x in the expected position, and ρ'' is the environment in which argument x is propagated to any function bound in *fmarks*. Function $\mathit{include}'$ adjusts the bodies of functions not being promoted and is defined as $\mathit{include}'\ (\_, \_, \mathit{fmarks}, x_0)\ (\rho : \mathit{pats}, \mathit{exp}) = (\rho : \mathit{pats}, \mathit{exp}[\rho''/\rho])$, where ρ'' is the environment in which the default value x_0 is used in applications of any function bound in *fmarks*.

5 Argument and Result Type Redefinition

Changes on the types of arguments or function result are denoted by labelling the corresponding changes in a signature changing clause.

If type t is labelled with l in the left-hand side of a signature changing clause, and the same label is used for type t' in the right-hand side of the same clause, then it is necessary to define a function of type $t \rightarrow t'$ to transform arguments in the *applications* of function f, and a function of type $t' \rightarrow t$ to transform the value of the corresponding argument of f wherever it is *used*. These transformation functions are defined by means of application and use clauses, respectively. In the application and use clauses of Section 3, each l_i is a label for the transformation

[7] Function MAP $f\ l$ applies function f to each element in list l, producing the list of results.

from type t_i to t'_i, e_i is an expression of type t_i, and e'_i is an expression of type t'_i; the corresponding transformation functions are $\lambda l_i.e'_i$ and $\lambda l_i.e_i$.

For example, another possible solution for the problem of including the *break* sequencer is to change the environment argument of semantic function $\mathcal{C}$ turning it into a pair: the first element represents the current environment, and the second element represents the continuation for *break* sequencers.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{transformation}\ \mathit{include\_break\_b}} \\
\quad\textbf{signature} & \mathcal{C}: & \mathit{Com} \to (r : \mathit{Env}) \to \mathit{Cc} \to \mathit{Cc} \\
 & & \qquad\quad \textbf{to}\ \mathit{Com} \to (r : (\mathit{Env}, \mathit{Cc})) \to \mathit{Cc} \to \mathit{Cc} \\
\quad\textbf{application} & \multicolumn{2}{l}{r \Rightarrow (r, \lambda s.\mathit{error})} \\
\quad\textbf{use} & \multicolumn{2}{l}{r \Rightarrow (\lambda(r', \_).r')\ r}
\end{array}
$$

The application of transformation *include_break_b* produces the following modified versions of the semantic equations of Figure 1[8]:

$$
\begin{array}{l}
\mathcal{C} : \mathit{Com} \to \underline{(\mathit{Env}, \mathit{Cc})} \to \mathit{Cc} \to \mathit{Cc} \\
\mathcal{P}[\![D;C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ \underline{(r, \lambda s.\mathit{error})})\ s_0 \\
\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r = \textsc{fix}\ \lambda f c.\mathcal{E}[\![E]\!]\ \underline{((\lambda(r', \_).r')r)}; \\
\qquad\qquad\qquad\qquad\qquad \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ (f\ c)\ \textbf{else}\ c.
\end{array}
$$

Sequencer break might then be defined by: $\mathcal{C}\ [\![\textbf{break}]\!]\ r\ (c, b) = b$.

5.1 Formal Aspects of Type Redefinition

Transformations to change types are collected, and the following structures are defined: $\beta = (\mathit{label}, \mathit{type}_1, \mathit{type}_2, \mathit{fmarks}, \mathit{app}, \mathit{use})$ is composed by the argument label, its original and new types, the list of all functions affected by the inclusion, and the application and use functions; *fmarks* may have the same structure as defined for argument inclusion, but can also represent a mapping from function names to $(\mathsf{return}, \mathit{type}^*)$, which represents the types of the function arguments. As it was the case with multiple inclusions, when multiple signature changes are performed on a function, they are sequentially handled so that each change consider the effects of the previous ones. For instance, the inclusion defined by transformation *include_break_b* is represented by $\beta_r = (r, \mathit{Env}, (\mathit{Env}, \mathit{Cc}), \mathit{fmarks}, \lambda r.(r, \lambda s.\mathit{error}), \lambda r.(\lambda(r', \_).r)\ r)$, where $\mathit{fmarks} = \{\mathcal{C} \mapsto (\mathsf{arg}, [\mathit{Com}], \mathit{Cc} \to \mathit{Cc})\}$.

Given a change $\beta = (x, t, t', \mathit{fmarks}, g, h)$ and specification environments τ and ρ, new environments τ' and ρ' are defined as:

$$
\begin{array}{l}
\tau' = \tau[t_1 \to \cdots \to t_k \to \underline{t'} \to t''/f_i, \forall(f_i \mapsto (\mathsf{arg}, [t_1, \cdots, t_k], t'')) \in \mathit{fmarks}] \\
\qquad\quad [t_1 \to \cdots \to t_k \to \underline{t'}/f_i, \forall(f_i \mapsto (\mathsf{return}, [t_1, \cdots, t_k])) \in \mathit{fmarks}] \\
\rho' = \{(f_i \mapsto \textsc{map}\ (\mathit{change}\ \beta)\ \mathit{clauses}) \mid \mathit{clauses} = \rho\ f_i\},
\end{array}
$$

[8] It is important to highlight that this version is still incomplete for the inclusion of the *break* sequencer requires the application of decorators.

Function *change* applies transformation β to each defining clause of a function, changing the environment of function applications to selectively interleave application or use functions in the definition, and is defined as:

$$change\ \beta\ (\rho : pats, exp) = (\rho : pats, exp[\rho''/\rho]),$$

where ρ'' maps each function f_j to a case depending on its relation with change β.

$$\begin{aligned}\rho'' = &\{(f_j \mapsto \text{MAP}\ (apply\ \beta)\ cs) \mid cs = \rho\ f_j, (\mathsf{arg}, t^*, t'') = fmarks\ f_j\}\\ &\bigcup\left\{(f_j \mapsto \text{MAP}\ (apply'\ \beta)\ cs) \mid cs = \rho\ f_j, (\mathsf{return}, t^*)\ = fmarks\ f_j\right\}\\ &\bigcup\{(f_j \mapsto \text{MAP}\ (use\ \beta\ \tau')\ cs) \mid cs = \rho\ f_j,\ f_j \text{ unbound in } fmarks\}\,.\end{aligned}$$

Function *apply* checks for argument conversions in promoted functions applying function g of β when necessary, and it is as follows:

$$apply\ (x, t, t', fmarks, g, h)\ (pats, exp) = (pats, \lambda\rho\ p_1 \cdots p_m x.exp\ \rho\ p_1\ \cdots\ p_m\ x'),$$

where $(p_1, \cdots, p_m)$ is a list of fresh identifiers corresponding to the list $(t_1, \cdots, t_m)$ bound to f in *fmarks*, and $x' = \textbf{if}\ typeof\ x = t\ \textbf{then}\ g\ x\ \textbf{else}\ x$. Function *apply′* checks for return conversions in promoted functions applying function g of β when necessary, and is defined as:

$$apply'\ (x, t, t', fmarks, g, h)\ (pats, exp) = (pats, g'\ exp),$$

where $g' = \lambda x.\textbf{if}\ typeof\ x = t\ \textbf{then}\ g\ x\ \textbf{else}\ x$. Function *use* is used in non-promoted functions, and checks the type of all their arguments of type t, applying function h when necessary. It looks as follows:

$$\begin{aligned}use\ (x, t, t', fmarks, g, h)\ \tau'\ ((\rho, p_1, \cdots, p_n), exp) =&\\ ((\rho, p_1, \cdots, p_n), exp[p'_1/p_1, \cdots, p'_n/p_n]),&\end{aligned}$$

where each $p'_i = \textbf{if}\ t_i = t\ \textbf{then}\ h'\ p_i\ \textbf{else}\ p_i$, $(t_1, \cdots, t_n)$ is the list of argument types bound to f in *fmarks*, and $h' = \lambda x.\textbf{if}\ typeof\ x = t'\ \textbf{then}\ h\ x\ \textbf{else}\ x$.

6 Function Decoration

Some language extensions may be simply implemented by changing the arguments passed to semantic functions. For instance, the definition of sequencer *break* may be included in the specification by adapting the environment in which the body of the *while* statement is executed. Decoration clauses define changes for arguments and result of a given semantic equation. In the decoration clauses of Section 3, each f_i is a function of type $t_{i1} \rightarrow t_{i2} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow t_i$, p_{ij} is a pattern for the j-th argument of f_i, and e_i is an expression of type t_i. The effect of this transformation is to replace all applications of function f_i matching the corresponding patterns by expression e_i.

For example, transformation *include_break_a* of Section 4 may be completed by the following decoration clause:

$$\textbf{replace}\ \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ b\ c\ \textbf{by}\ \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ c\ c$$

This decoration clause only affects the equation defining the *while* statement shown in Section 4, and its modified version is:

$$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ b\ c = (\text{FIX}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ c\ (f\ c')\ \textbf{else}\ c')\ c.$$

6.1 Formal Aspects of Function Decoration

Transformations for function decoration and equation redefinitions are collected, and the following structure is defined: $\gamma = (\textit{function-name}, \textit{patterns}, \textit{expression})$, which comprises the name of the function being decorated or redefined, the applicable patterns for the function, and the corresponding new expression. As it was the case with multiple inclusions, when multiple decorations and redefinitions are performed on a function, they are sequentially handled so that each change consider the effect of the previous ones. For instance, the decoration clause of transformation *include_break_a* is represented by $\gamma_{\mathcal{C}} = (\mathcal{C}, [[\![\textbf{while}\ E\ C]\!], r, b, c], \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ c\ c)$.

Given a decoration clause $\gamma = (f_i, \textit{pats}, \textit{exp})$ and a specification environment ρ, a new environment ρ' is defined as $\rho' = \rho[(\text{MAP}\ (\textit{decorate}\ \gamma)\ \textit{clauses})/f_i]$, where $\textit{clauses} = \rho\ f_i$. Function *decorate* takes as argument the decoration clause γ and the function clauses, and defines a decorated version of the function, considering all function applications in *exp* be related to the old version of the function. This function is defined as: $\textit{decorate}\ (f, (p_1, \cdots, p_m), \textit{exp})\ ((p'_1, \cdots, p'_n), \textit{exp}') = \textit{clause}'$, where if $(p_1, \cdots, p_m)$ is a generalization[9] of $(p'_1, \cdots, p'_m)$, then

$$\begin{aligned} \textit{clause}' &= ([p_1, \cdots, p_m, p'_{m+1}, \cdots, p'_n], \textit{exp}''\ p'_{m+1} \cdots p'_n), \\ \textit{exp}'' &= \textit{exp}[\textit{exp}'/f][p'_1/p_1, \cdots, p'_m/p_m], \end{aligned}$$

or $\textit{clause}' = ((p'_1, \cdots, p'_n), \textit{exp}')$, otherwise.

The effect of function *decorate* is to replace each occurrence of f in the environment by its new version, in which whenever the arguments of the application match $p_1, \cdots, p_m$, the decorated expression replaces the original function application. If the patterns do not match, the original definition of f is used in the application. It is important to highlight that all free occurrences of f in *exp* are replaced by an application of $\textit{exp}'$, and for this reason any application of function f refers to the *original definition* of f.

[9] Pattern p is a generalization of pattern p' if all expressions matching p' also matches p.

7 Equation Redefinition

In some situations, it may be more appropriate to rewrite the definition of some constructs by means of a redefinition clause than to adapt the existing equations. In the redefinition clauses, each f_i is a function of type $t_{i1} \rightarrow t_{i2} \rightarrow \cdots \rightarrow t_{ik_i} \rightarrow t_i$, p_{ij} is a pattern for the j-th argument of f_i, and e_i is an expression of type t_i. For instance, the *while* statement could be redefined as:

$$\begin{array}{l} \textbf{transformation}\ \mathit{include\_break\_d} \\ \quad \textbf{redefine}\ \mathcal{C}\ [\![\textbf{while}\ E\ C]\!]\ r\ = \\ \qquad\qquad \textsc{fix}\ \lambda f c.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r[c/\mathsf{break}]\ (f\ c)\ \textbf{else}\ c \end{array}$$

7.1 Formal Aspects of Equation Redefinition

Transforming redefinition clauses is similar to transforming decoration clauses. Given a redefinition clause $\gamma = (f_i, \mathit{pats}, \mathit{exp})$ and a specification environment ρ, a new environment ρ' is defined as $\rho' = \rho[\textsc{map}\ (\mathit{redefine}\ \gamma)\ \mathit{clauses}/f_i]$, where $\mathit{clauses} = \rho\ f_i$. Function *redefine* takes as argument the redefinition clause γ and the function clauses, and replaces the matching clauses. This function is defined as[10] $\mathit{redefine}\ (f, (p_1, \cdots, p_m), \mathit{exp})\ ((p'_1, \cdots, p'_n), \mathit{exp}') = \mathit{clause}'$, where if $(p_1, \cdots, p_m)$ is a generalization of $(p'_1, \cdots, p'_m)$ then

$$\begin{array}{ll} \mathit{clause}' & = ([p_1, \cdots, p_m, p'_{m+1}, \cdots, p'_n], \mathit{exp}''\ p'_{m+1} \cdots p'_n), \\ \mathit{exp}'' & = \mathit{exp}[p'_1/p_1, \cdots, p'_m/p_m], \end{array}$$

or $\mathit{clause}' = ((p'_1, \cdots, p'_n), \mathit{exp}')$, otherwise.

The effect of function *redefine* is to replace each occurrence of f in the environment by its new version, in which whenever the arguments of the application match $p_1, \cdots, p_m$, the new expression replaces the original function application. If the patterns do not match, the original definition of f is used in the application. It is important to highlight that all free occurrences of f in *exp* are looked up to in the current environment, and for this reason any application of function f refers to the *new definition* of f.

8 Case Study: Definition of Procedures and Advices

Aspect-oriented concepts of advices and dynamic join points [Kiczales et al. 1997] are formally defined by Wand, Kiczales, and Dutchyn in [Wand et al. 2004], who present a monadic denotational semantic for a simple functional language ressembling Scheme and composed by global procedures, advices and pointcut description. The semantic equations presented in their

[10] Compare with the definition of *replace*, which applies the original version of the function.

Sets:

v	$\in Val$	Expressed values
l	$\in Loc$	Locations
s	$\in Sto$	Stores
id	$\in Id$	Identifiers
$pname$, $wname$	$\in Pname$	Procedure names
v	$\in Val$	Expressed values

Join points, pointcut designators:

$$\begin{array}{lll} jp & \in & JP \\ jp & \rightarrow & \langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle \\ k & \rightarrow & \mathsf{pcall} \mid \mathsf{pexecution} \mid \mathsf{aexecution} \\ pcd & \rightarrow & \cdots \end{array}$$

Execution monad:

$$T(A) = JP \times Sto \rightarrow (A \times Sto)_\perp$$

Semantic Domains:

$\pi \in Proc = Val^* \rightarrow T(Val)$	Procedures
$\alpha \in Adv = JP \rightarrow Proc \rightarrow Proc$	Advices
$\phi \in PE = Pname \rightarrow Proc$	Procedure environments
$\gamma \in AE = Adv^*$	Advice environments
$\rho \in Env = [Id \rightarrow Loc]$	Environments

Figure 2: Working Example – Basic Definitions for the Semantics of Aspect-Oriented Advices and Join Points (Taken from [Wand et al. 2004])

specification contain interleaved elements which makes it hard to fully understand the key concepts of the definition. This paper simplifies that formalization by applying the introduced mechanisms of incremental specification. This case study is a first step to the validation of the proposed technique. The presented version does not consider *within* and *proceed* clauses, which can be straightfowardly included by means of environment decorations.

Figure 2 summarizes key features of the semantic specification, namely its main domains and execution monad, which were taken *ipsis litteris* from the original paper [Wand et al. 2004]. That paper also contains details on the algebra of pointcuts, auxiliary functions, and monad operations, advised to readers looking foward to deeply understand the formalization. Procedures are the starting point of the definition. Procedure declarations are defined by function $\mathcal{P}$, which creates an procedure environment which associates the procedure name with the corresponding procedure semantics:

$$\begin{array}{l} \mathcal{P} : Procedure \rightarrow PE \rightarrow PE \\ \mathcal{P}[\![(\textbf{procedure}\ pname\ (x_1, \cdots, x_n)\ e)]\!]\ \phi = [proc/pname] \\ \text{where } proc = \lambda(v_1, \cdots, v_n). \textbf{let}\ l_1 \Leftarrow alloc\ v_1; \cdots; l_n \Leftarrow alloc\ v_n \\ \qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ \mathcal{E}[\![e]\!]\ [l_1/x_1, \cdots, l_n/x_n]\ \phi \end{array}$$

Procedure calls are defined by means of function $\mathcal{E}$, which evaluates the arguments and applies to it the procedure bound in the environment:

$$\begin{array}{l} \mathcal{E} : Exp \rightarrow Env \rightarrow PE \rightarrow T(Val) \\ \mathcal{E}[\![(pname\ e_1\ \cdots\ e_n)]\!]\ \rho\ \phi = \textbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\ \rho\ \phi; \cdots; v_n \Leftarrow \mathcal{E}[\![e_n]\!]\ \rho\ \phi \\ \qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ \phi\ pname\ (v_1, \cdots, v_n) \end{array}$$

The first step to include aspect-oriented features consists of defining procedures to depend on advice environments, which is solved by means of trans-

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{transformation}\ \textit{include-advices}} \\
\quad\textbf{signature} & \mathcal{E} : \textit{Exp} \to \textit{Env} \to \textit{PE} \to T(\textit{Val}) \\
& \qquad\quad \textbf{to}\ \textit{Exp} \to \textit{Env} \to \textit{PE} \to (\gamma : \textit{AE}) \to T(\textit{Val}), \\
& \mathcal{P} : \textit{Procedure} \to \textit{PE} \to \textit{PE}\ \textbf{to}\ \textit{Procedure} \to \textit{PE} \to (\gamma : \textit{AE}) \to \textit{PE} \\
\quad\textbf{default} & \gamma = [] \\
\quad\textbf{replace} & \underline{\mathcal{P}[\![(\textbf{procedure}\ \textit{pname}\ (x_1, \cdots, x_n)\ e)]\!]\ \phi\ \gamma} \\
& \qquad \textbf{by}\ [(\textit{enter-jp}\ \gamma\ (\textit{new-pexecution pname})\ \textit{proc})/\textit{pname}] \\
& \qquad\qquad \text{where}\ \textit{proc} = \underline{\mathcal{P}[\![(\textbf{procedure}\ \textit{pname}\ (x_1, \cdots, x_n)\ e)]\!]\ \phi\ \gamma}, \\
& \underline{\mathcal{E}[\![(\textit{pname}\ e_1\ \cdots\ e_n)]\!]\ \rho\ \phi\ \gamma} \\
& \qquad \textbf{by}\ \underline{\mathcal{E}[\![(\textit{pname}\ e_1\ \cdots\ e_n)]\!]\ \rho\ (\phi[\textit{proc}/\textit{pname}])\ \gamma} \\
& \qquad\qquad \text{where}\ \textit{proc} = \lambda v^*.\textit{enter-jp}\ \gamma\ (\textit{new-pcall pname}\ v^*)\ (\phi\ \textit{pname})
\end{array}
$$

Figure 3: Transformation Function for Including Advices in the Specification

formation *include-advices* of Figure 3. This transformation: (a) includes advice environment as argument of functions $\mathcal{E}$ and $\mathcal{P}$ by means of a signature change clause; such environment is propapagated through recursive calls of function $\mathcal{P}$, and in the absence of an advice environment the default empty environment is used; (b) decorates the procedure environment to weave execution advices, by means of the first replace clause, which decorates the result of function $\mathcal{P}$ with specific joinpoint marks; and (c) decorates the execution of procedure call expressions to weave execution advices, by means of the second replace clause, which decorates the procedure environment argument of function $\mathcal{E}$ with specific joinpoint marks.

The semantics of advices are given by function $\mathcal{A}$, which executes the advice and the procedure bodies in the correct order, if the corresponding pointcut is applicable to the procedure; otherwise, only the procedure body is executed.

$$
\begin{array}{l}
\mathcal{A} : \textit{Advice} \to \textit{PE} \to \textit{AE} \to \textit{JP} \to \textit{Proc} \to \textit{Proc} \\
\mathcal{A}[\![(\textbf{before}\ \textit{pcd}\ e)]\!]\ \phi\ \gamma\ \textit{jp}\ \pi = \\
\qquad \lambda v^*.\mathcal{PCD}[\![\textit{pcd}]\!]\ \textit{jp}\ (\lambda\rho.\textbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e]\!]\ \rho\ \phi\ \gamma; v_2 \Leftarrow \pi\ v^*\ \textbf{in}\ v_2)\ (\pi\ v^*) \\
\mathcal{A}[\![(\textbf{after}\ \textit{pcd}\ e)]\!]\ \phi\ \gamma\ \textit{jp}\ \pi = \\
\qquad \lambda v^*.\mathcal{PCD}[\![\textit{pcd}]\!]\ \textit{jp}\ (\lambda\rho.\textbf{let}\ v_1 \Leftarrow \pi\ v^*; v_2 \Leftarrow \mathcal{E}[\![e]\!]\ \rho\ \phi\ \gamma\ \textbf{in}\ v_1)\ (\pi\ v^*)
\end{array}
$$

By applying the proposed transformation techniques, one can provide vague definition of procedures which can be extended to support advices by means of transformations.

9 Conclusions

This paper presented a new approach to improve the modularity of denotational semantic specifications. This approach, based on the vagueness properties of initial definitions and on their transformations, may improve the readability of semantic equations by separating the concerns on interfering language constructs. In an incremental definition, it is possible to include new constructs without rewriting previously written equations, even in those cases where one construct must prepare the context for others. The objective of the proposed model is to permit that constructs be presented in a more intuitive way, which can be closer to their usual natural language descriptions: each construct may be isolated for better comprehension.

The proposed methodology for incremental definition provides simple mechanisms for the definition of programming languages semantics. The definition of a language construct may be vague with regard to other constructs, and then it becomes easier to understand its semantics, because the relationship among constructs is separately defined and does not get in the way. Although vagueness is an essential feature in informal definitions, there is no way of controlling it, and incomplete definitions may look like vague ones. When applied to formal definitions, vagueness helps constructing the bond to informal definitions, leading to a better readability. Furthermore, this technique can be used along with other modularity approaches for denotational semantics, such as monads, benefiting from their positive aspects.

A very difficult problem to handle in denotational semantics transformation is to preserve two essential properties: irrelevance of definition order and abstractness. The proposed methodology defines a recommended reading order for the equations, because each specification could be interpreted as a chapter of the language manual. However, future tool support should provide a way of generating the complete set of equations for any intermediate specifications, by weaving transformation code into the existing equations.

Abstractness of denotational semantics is also preserved because all transformations only require textual substitution to generate the woven specification. Thus, transformation functions do not break abstractness of existing equations, so that the denotation of a construct remains dependent only on the denotation of its constituents and the current context as expected.

Furthermore, extensibility is improved by the separation of concerns provided by vagueness in specifications. Higher degrees of extensibility are only achieved in software systems when modules are simple and independent, because it becomes easier to cope with modifications. Modules defined by means of the proposed approach tend to handle minimal information, with direct impact on the overall modularity quality.

Future work comprehends the investigation of techniques for implementing the proposed constructs and further studies of its properties and consequences. Although the transformations presented in this paper apply on the denotations of language constructs, the authors believe that it can be implemented on top of general term-based rewriting systems, such as Maude [Clavel et al. 2003]. Full validation of the proposed model is under development. Because of the inherent complexity of large scale programming languages, the proposed approach does not completely solve the scalability problem of denotational semantics, but represents an important step forward in the direction of simplifying the practical use of this method.

Acknowledgments

This work is partially supported by CAPES (grant 3680-06-1), Fapemig (grant 11740), and the Portuguese Science Foundation (FCT) (grant SFRH/BSAB/782/2008). The authors would like to thank professor Peter D. Mosses, from Swansea University, for his valuable suggestions on a previous version of this paper, as well as the anonymous referees whose insights were very helpful in the production of the paper's final version.

References

[Cenciarelli et al. 1999] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. *Lecture Notes in Computer Science*, 1523:157–200, 1999.

[Clavel et al. 2003] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, 2003.

[Moor et al. 2000] O. de Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica*, 24(3), 2000.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, page 220ff. Springer-Verlag, 1997.

[Liang et al. 1995] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995.

[Moggi 1991] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mosses 1977] P. D. Mosses. Making denotational semantics less concrete. In *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, number 41 in Bericht, pages 102–109. Abteilung Informatik, Universität Dortmund, 1977.

[Mosses 2004] P. D. Mosses. Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.

[Mosses 2005] P. D. Mosses. A constructive approach to language definition. *Journal of Universal Computer Science*, 11(7):1117–1134, 2005.

[Wand et al. 2004] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[Wansbrough and Hamer 1997] K. Wansbrough and J. Hamer. A Modular Monadic Action Semantics. In *Proceedings of the Conference on Domain-Specific Languages, Santa Barbara, California*, pages 157–170. The USENIX Association, 1997.