Disentangling Denotational Semantics Definitions

Fabio Tirelo[1], Roberto S. Bigonha[2], Jo˜ao A. B. Saraiva[3]

[1]Instituto de Informática – Pontificia Universidade Católica de Minas Gerais
Belo Horizonte, MG, Brazil

[2]Departamento de Ciência da Computaç˜ao – Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brazil

[3]Departamento de Informática – Universidade do Minho
Braga, Portugal

ftirelo@pucminas.br, bigonha@dcc.ufmg.br, jas@di.uminho.pt

Abstract. Denotational semantics is a powerful technique for the formal definition of programming languages. However, because language constructs are not always orthogonal, usually many semantic equations in a definition must be aware of unrelated constructs semantics. Current modularity approaches for this formalism do not address this problem, providing for this reason tangled semantic definitions. This paper proposes an incremental approach for denotational semantics, 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.

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 impose 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 found in 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 traditional approaches of denotational semantics 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 se-

mantics. 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, having then 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 improve the modularity of denotational semantics have been made since then, with highlight to Monadic Semantics [Liang et al. 1995, Moggi 1991, Wadler 1990] and Monadic Action Semantics [Wansbrough and Hamer 1997]. Recently problems related to separation of concerns in semantic specifications were addressed in [de Moor et al. 2000, 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.

An aspect-oriented based technique to improve the modularity of attribute grammars, which can also be applied to semantics definitions, has been proposed by [de Moor et al. 2000]. In that model, attributes may be defined in separate sections and

[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.

"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 repeti-

tion 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 instance, 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 { ... } }
```

In [Mosses 2004], a model inspired on monadic semantics for defining modular structural operational semantics (MSOS) of programming languages is proposed. 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. In addition, as SOS allows one to separately define 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 in [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].

Reuse degree can also be improved by means of the constructive approach proposed in [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. In this approach, 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 semantics 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; 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 from the apprentice to learn the language concepts.

Let a semantics specification be the quaduple $S = (G, D, \tau, \varrho)$, where G is the language abstract grammar, D is the set of semantic domains, τ is a type environment

[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.

Page 4

which maps semantic functions into their domains, and ϱ is the environment which maps semantic functions 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 called by means of 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] $([\varrho, p_{i1}, \cdots, p_{ik}], e_i[(\varrho\ f_j)\ \varrho/f_j, \forall j])$. 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 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 E : Exp → Env → Val, such that E[[Id]] r = r Id, and $E[[E_1 + E_2]]\ r = E[[E_1]]\ r + E[[E_2]]\ r$, then type environment τ is defined as $\tau = \{E \mapsto\!\!\!\rightarrow (Exp \rightarrow Env \rightarrow Val)\}$, and definition environment ϱ is defined as $\varrho = \{E \mapsto\!\!\!\rightarrow \{clause_1, clause_2\}\}$, where $clause_1 = ([\varrho, [[Id]], r], r\ Id)$ and $clause_2 = ([\varrho, [[E_1 + E_2]], r], ((\varrho\ E)\ \varrho)\ [[E_1]]\ r + ((\varrho\ E)\ \varrho)\ [[E_2]]\ r)$.

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 $S_{i+1} = t(S_i) + \Delta S_i$ may be 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 mapping 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, τ, ϱ)=(G, D, τ′, ϱ′). An inclusion into a specification S, denoted by ΔS, represents new elements to be added in the specification, usually elements concerning 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 addapt the while equation to prepare the context in which the sequencer is executed. The corresponding inclusions consist of 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

[3] In expression ϱ f, consider f be the function identifier and not the actual function itself.

[4] FIX represents the fix point operator.

Abstract syntax:

P ∈ Prog → D;C
C ∈ Com → while E C | C_1;C_2 | ⋯
E ∈ Exp →⋯
D ∈ Dec →⋯

Semantic Functions:

P : Prog → Ans
C : Com → Env → Cc → Cc
E : Exp → Env → (Val → Cc) → Cc
D : Dec → Env → (Env → Cc) → Cc

Semantic Equations:

P[[D;C]] = D[[D]] r_0 (λr.C[[C]] r (λs.stop)) s_0
C[[C_1;C_2]] r c = C[[C_1]] r; C[[C_2]] r c
C[[while E C]] r = FIX λfc.E[[E]] r; λv.if v then C[[C]] r (f c) else c
Other equations defining C, E, and D

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

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) function 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 are performed before equation redefinitions.

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

[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].

Page 6

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 C representing the continuation for a break statement, as defined by transformation include break a.

transformation include break a
signature C : Com → Env → Cc → Cc
to Com → Env → (b : Cc) → Cc → Cc
default b = λs.error

To every application of function C a new third argument λs.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]:

C : Com → Env → Cc → Cc → Cc
P[[D;C]] = D[[D]] r_0 (λr.C[[C]] r (λs.error) (λs.stop)) s_0
C[[C_1;C_2]] rbc = C[[C_1]] r b; C[[C_2]] rbc
C[[while E C]] rbc = (FIX λfc .E[[E]] r; λv.if v then C[[C]] r b (f c) else c) c

Sequencer break might then be defined by: C [[break]] rbc = b.

4.1. Formal Aspects of Argument Inclusion

Transformations for argument inclusions are collected, and the following structures are defined: α = (label,type,fmarks,def-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; fmarks = function-name → (arg,$type_*$,type) maps function names into a tuple composed by a list of types, corresponding to the arguments after which the new argument is included, and the result type of the function. When multiple argument inclusions are performed on a function, they are sequentially handled so that each inclusion consider the effect of the previous ones. For instance, the inclusion defined by transformation include break a is represented by α_b = (b,Cc,fmarks, λs.error), where fmarks = {C ↦→ (arg,[Com,Env],Cc → Cc)}.

[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 and requires the application of decorators.

Given an inclusion $\alpha = (x, t, fmarks, x_0)$ and specification environments τ and ϱ, new environments τ and ϱ are defined as[7]:

$$\tau = \tau[t_1 \rightarrow \cdots \rightarrow t_k \rightarrow t \rightarrow t \,/ f_i, \forall (f_i \mapsto\!\!\rightarrow (arg, [t_1, \cdots, t_k], t)) \in fmarks]$$

$$\varrho = \{(f_i \mapsto\!\!\rightarrow \text{MAP (include } \alpha) \text{ clauses}) \mid clauses = \varrho\ f_i, f_i \text{ bound in fmarks}\}$$
$$\cup \{(f_i \mapsto\!\!\rightarrow \text{MAP (include } \alpha) \text{ clauses}) \mid clauses = \varrho\ f_i, f_i \text{ unbound in fmarks}\}$$

Function include performs the inclusion of the argument in the functions being promoted and is defined as include (x, ,fmarks,) (ϱ : pats,exp)=(ϱ : pats ,exp[ϱ /ϱ]), 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 include adjusts the bodies of functions not being promoted and is defined as include (, ,fmarks,x_0) (ϱ : pats,exp) = (ϱ : pats,exp[ϱ /ϱ]), 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 → t to transform arguments in the applications of function f, and a function of type t → 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 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 to the problem of including the break sequencer is to change the environment argument of semantic function C to be a pair: the first element represents the current environment, and the second element represents the continuation for break sequencers.

```
transformation include break b
    signature       C :     Com → (r : Env) → Cc → Cc
                                to Com → (r : (Env,Cc)) → Cc → Cc
    application r ⇒ (r, λs.error)
    use             r ⇒ (λ(r , ).r ) r
```

The application of transformation include break b produces the following modified versions of the semantic equations of Figure 1[8]:

C : Com → (Env,Cc) → Cc → Cc

P[[D;C]] = D[[D]] r_0 (λr.C[[C]] (r, λs.error)) s_0

C[[while E C]] r = FIX λfc.E[[E]] ((λ(r ,).r)r); λv.if v then C[[C]] r (f c) else c.

Sequencer break might then be defined by: C [[break]] r (c, b) = b.

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

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

Page 8

5.1. Formal Aspects of Type Redefinition

Transformations to change types are collected, and the following structures are defined: $\beta = (label, type_1, type_2, fmarks, app, use)$ 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 $(return, type_*)$, 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 effect of the previous ones. For instance, the inclusion defined by transformation include break b is represented by $\beta_r = (r, Env, (Env, Cc), fmarks, \lambda r.(r, \lambda s.error), \lambda r.(\lambda(r\ ,\ ).r)\ r)$, where fmarks = $\{C \mapsto\!\!\!\rightarrow (arg, [Com], Cc \rightarrow Cc)\}$.

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

$$\tau = \tau[t_1 \rightarrow \cdots \rightarrow t_k \rightarrow t \rightarrow t\ /f_i, \forall(f_i \mapsto\!\!\!\rightarrow (arg, [t_1, \cdots, t_k], t)) \in fmarks]$$
$$[t_1 \rightarrow \cdots \rightarrow t_k \rightarrow t\ /f_i, \forall(f_i \mapsto\!\!\!\rightarrow (return, [t_1, \cdots, t_k])) \in fmarks]$$
$$\varrho = \{(f_i \mapsto\!\!\!\rightarrow \text{MAP (change } \beta) \text{ clauses}) \mid clauses = \varrho\ f_i\},$$

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:

$$\text{change } \beta\ (\varrho : pats, exp) = (\varrho : pats, exp[\varrho\ /\varrho]),$$

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

$$\varrho = \{(f_j \mapsto\!\!\!\rightarrow \text{MAP (apply } \beta) \text{ clauses}) \mid clauses = \varrho\ f_j, (arg, t_*, t) = fmarks\ f_j\}$$
$$\cup\ \{(f_j \mapsto\!\!\!\rightarrow \text{MAP (apply } \beta) \text{ clauses}) \mid clauses = \varrho\ f_j, (return, t_*) = fmarks\ f_j\}$$
$$\cup\ \{(f_j \mapsto\!\!\!\rightarrow \text{MAP (use } \beta\ \tau) \text{ clauses}) \mid clauses = \varrho\ f_j,\ f_j \text{ unbound in } fmarks\}.$$

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

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

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 = if typeof x = t then g x 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 = λx.if typeof x = t then g x 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 is defined as:

use (x, t, t ,fmarks, g, h) τ ((ϱ, $p_1,\cdots,p_n$),exp) =
((ϱ, $p_1,\cdots,p_n$),exp[$p_1/p_1,\cdots,p_n/p_n$]),

where each p_i = if t_i = t then h p_i else p_i, $(t_1,\cdots,t_n)$ is the list of argument types bound to f in fmarks, and h = λx.if typeof x = t then h x else x.

Page 9

6. Function Decoration

Some language additions 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:

replace C [[while E C]] rbc by C [[while E C]] rcc

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

C[[while E C]] rbc = (FIX λfc .E[[E]] r; λv.if v then C[[C]] r c (f c) 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: γ = (function-name,patterns,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 γ_C = (C,[[[while E C]], r, b, c],C [[while E C]] rcc).

Given a decoration clause $\gamma = (f_i,pats,exp)$ and a specification environment ϱ, a new environment ϱ is defined as $\varrho = \varrho[(\text{MAP (decorate } \gamma) \text{ clauses})/f_i]$, where clauses = ϱ 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: decorate $(f,(p_1,\cdots,p_m),exp)$ $((p_1,\cdots,p_n),exp)$ = clause , where if $(p_1,\cdots,p_m)$ is a generalization[9] of $(p_1,\cdots,p_m)$, then

$$\begin{aligned} \text{clause} &= ([p_1,\cdots,p_m,p_{m+1},\cdots,p_n],\text{exp } p_{m+1}\cdots p_n), \\ \text{exp} &= \text{exp}[\text{exp}/f][p_1/p_1,\cdots,p_m/p_m], \end{aligned}$$

or clause = $((p_1,\cdots,p_n),exp)$, 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 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:

transformation include break d

redefine C [[while E C]] r =

FIX λfc.E[[E]] r; λv.if v then C[[C]] r[c/break] (f c) else c

7.1. Formal Aspects of Equation Redefinition

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

$$clause = ([p_1,\cdots,p_m,p_{m+1},\cdots,p_n], exp\ p_{m+1}\cdots p_n),$$
$$exp = exp[p_1/p_1,\cdots,p_m/p_m],$$

or $clause = ((p_1,\cdots,p_n),exp')$, 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: Incremental Definition of Procedures and Advices

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

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

Sets:

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

Join points, pointcut designators:

jp	∈ JP	
jp	→ ⟨⟩ | ⟨k,pname,wname,v_*,jp⟩	
k	→ pcall | pexecution | aexecution	
pcd	→ ⋯	

Execution monad:

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

Semantic Domains:

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

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

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 P, which creates an procedure environment which associates the procedure name with the corresponding procedure semantics:

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

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

$$\begin{aligned}
&E : Exp \to Env \to PE \to T(Val)\\
&E[[(\text{pname } e_1 \cdots e_n)]]\ \varrho\ \phi = \text{let } v_1 \Leftarrow E[[e_1]]\ \varrho\ \phi;\cdots;v_n \Leftarrow E[[e_n]]\ \varrho\ \phi\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{in } \phi\ \text{pname } (v_1,\cdots,v_n)
\end{aligned}$$

The first step to include aspect-oriented features consists of defining procedures to depend on advice environments, which is solved by means of transformation include-advices of Figure 3. This transformation: (a) includes advice environment as argument of functions E and P by means of a signature change clause; such environment is propapagated through recursive calls of function 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 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 E with specific joinpoint marks.

The semantics of advices are given by function A, which executes the advice and the procedure bodies in the correct order, if the corresponding pointcut is applicable to

Page 12

transformation include-advices
signature E : Exp → Env → PE → T(Val)
to Exp → Env → PE → (γ : AE) → T(Val),
P : Procedure → PE → PE to Procedure → PE → (γ : AE) → PE
default γ = []
replace P[[(procedure pname (x_1,⋯,x_n) e)]] ϕ γ
by [(enter-jp γ (new-pexecution pname) proc)/pname]
where proc = P[[(procedure pname (x_1,⋯,x_n) e)]] ϕ γ,
E[[(pname e_1 ⋯ e_n)]] ϱϕγ
by E[[(pname e_1 ⋯ e_n)]] ϱ (ϕ[proc/pname]) γ
where proc = λv_*.enter-jp γ (new-pcall pname v_*) (ϕ pname)

Figure 3. Transformation Function for Including Advices in the Specification

the procedure; if the pointcut is not applicable, then only the procedure body is executed.

A : Advice → PE → AE → JP → Proc → Proc
A[[(before pcd e)]] ϕ γ jp π =
λv_*.PCD[[pcd]] jp (λϱ.let v_1 ⇐ E[[e]] ϱϕγ;v_2 ⇐ π v_* in v_2) (π v_*)
A[[(after pcd e)]] ϕ γ jp π =
λv_*.PCD[[pcd]] jp (λϱ.let v_1 ⇐ π v_*;v_2 ⇐ E[[e]] ϱϕγ in v_1) (π v_*)

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 semantics specifications. This solution, 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.

Page 13

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] and Stratego [Visser 2004]. 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.

10. Acknowledgments

This work is partially supported by Capes and Fapemig. 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, P., Knapp, A., Reus, B., and Wirsing, M. (1999). An event-based structural operational semantics of multi-threaded Java. Lecture Notes in Computer Science, 1523:157–200.

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

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

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

Liang, S., Hudak, P., and Jones, M. (1995). Monad Transformers and Modular Inter-

preters. In POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 333–343, New York, NY, USA.

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

Mosses, P. D. (1977). 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.

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

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

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C. et al., editors, Domain-Specific Program Generation, volume 3016 of Lecture Notes in Computer Science, pages 216–238. Spinger-Verlag.

Wadler, P. (1990). Comprehending Monads. In LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming, pages 61–78, New York, NY, USA. ACM Press.

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

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