The Solution to the Scalability Problem of Denotational Semantics

Fabio Tirelo[1], **Roberto da Silva Bigonha**[2]

[1]Instituto de Informática – Pontifícia Universidade Católica de Minas Gerais
Av. Dom José Gaspar, 500 – Coração Eucaristíco
30535-610 – Belo Horizonte – MG – Brazil

[2]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Av. Presidente Antônio Carlos, 6627 – Campus Pampulha
31270-901 – Belo Horizonte – MG – Brazil

ftirelo@pucminas.br, bigonha@dcc.ufmg.br

Abstract. *Denotational semantics is a powerful technique for the formal definition of programming languages. However, inherent problems of large scale languages arise difficulties not yet addressed by current approaches. This paper categorizes a source of the scalability problem of denotational semantics, and presents a solution based on context transmission and module transformations.*

1. Introduction

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

Denotational semantics of programming languages [Scott and Strachey 1971] defines the semantics of each construct as its contribution to the program final answer, and usually is presented as a function from its context to an answer [Wadsworth 1978]. For instance, for languages having variables and assignments, the context may be composed by stores and environments; the presence of sequencers requires that context be additionally composed by continuations. In an incremental definition, some context information should be only identified when it is necessary to define some construct; not anticipating such problem usually causes the redefinition of previously defined modules, so that the whole writing/rewriting process becomes a tedious and error-prone activity.

Current approaches for the definition of modular denotational semantics are based upon the abstraction of context expression [Tirelo 2005]. For instance, Wadler [Wadler 1990] suggests stores, environments, and continuations be implicitly transmitted by means of monads [Moggi 1991]. Thus, constructs not directly using variables may not directly reference current store and environment. Simplifications of the same nature may be applied in equations not involving the direct use of continuations.

However, large scale programming languages have features which cannot be completely understood when isolated from other language features, since they can be mutually interfering. This property directly impacts the modularity of denotational semantics

descriptions, since the description of one construct must contain elements of related constructs, which violates the principle of module high cohesion [Meyer 1997]. This problem, explained in details in Section 2, is not adequately addressed in current denotational semantics methodologies.

One of the first steps to the modularity of denotational semantics has been made by Peter Mosses in the Action Semantics [Mosses 1977, Mosses 1992]. 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]. This work improves the results of those contributions by presenting a modular mechanism for defining and transmitting context information among programming languages contructs. The Notus language (see Section 4) has been inspired on traditional denotational semantics notation [Mosses 1979, Scott and Strachey 1971]. It differs from those works by allowing syntax and semantics to be defined in an incremental way by means of extensions and module transformers (see Section 6).

Module transformation has been inspired in monad transformers [Liang et al. 1995] and in the weaving process of aspect-oriented programming [Kiczales et al. 1997, Wand et al. 2004]. The main difference between monad and module transformers is that the latter may be selectively applied in different function calls, and can be ignored when necessary. The weaving process of aspect-oriented programming usually defines an auxiliary language to describe pointcuts and advices; Notus uses no additional concept to define weaving, and only traditional denotational semantics functions are necessary. Modular context transmission (see Section 5) has been inspired in the Wormhole Pattern implemented in AspectJ [Laddad 2003], whose purpose is to transmit information from a caller module to worker modules in complex tree-based object structures. By letting the transmission implicit in function calls involving contexts, language designers do not need to define the protocol of transmission, which usually leads to untangled definitions.

Section 3 shows how the main current approaches for modularity of denotational semantics addresses the problem presented in Section 2. Sections 4, 5, and 6 define a new approach for denotational semantics modularity, which comprehends a technique for modular context manipulation. Section 7 presents a qualitative evaluation of the presented approach and contains the final considerations.

2. Contexts in Programming Language Semantics

According to [Parnas 1972], modularity degree in software systems depends on the chosen decomposition technique. Denotational semantics specifications consist of defining the meaning of each construct of the language's abstract syntax, so that this must be the dominant decomposition rule. However, there are language features, such as dynamic type checking, which are difficult to modularize. These difficulties are present in the form of tangling and scattering, whose solution by means of aspect-oriented programming is presented by [Kiczales et al. 1997].

The main reason for this difficulty is the fact that usual features in programming languages, such as dynamic type checking, error handling, and sequencers, influence the behavior of others, and such influence must be explicit in their defining semantic equa-

tions. Therefore, language constructs cannot be fully understood when isolated, because their meanings depend on their context and on general language properties. According to [Tirelo 2005, Bigonha 2006], besides its constituents, a denotational semantics of some language construct must consider the following context components: (i) its *antecedents*; (ii) its possible *destinations*; (iii) and its *involving structure*. Moreover, context information derived from *general properties in the language design* must be taken into account, so their enforcement appears in the final semantic equations.

The *antecedents* of a construct comprehend the effects of previous execution steps and are usually propagated by means of stores and environments. For instance, the antecedents of an expression evaluation contain the mapping of each variable used in the expression into the last value assigned to it.

The *destination* of a construct consists of the next step in the execution, and is usually represented by means of continuations. Some constructs may have more than one possible destination, which are usually propagated through the environment. As an example of multiple destinations of a construct, consider the denotation of an arbitrary command C in an imperative language such as Java. For such construct, there must be context propagation associated with the next step of C, to be used when C does not contain sequencers, and another to be used if any of the following sequencers occur in C: *break*, *continue*, *return*, and *throw*.

The *inclosing structure* consists of the construct context in the abstract syntax tree of a program. For instance, to define the semantics of the break statement in Java, it is necessary to define if it is inside a repetition statement or inside a switch-case; in addition, in the case of nested statements, it is necessary to decide which statement the break interrupts. This problem is also found in the definition of exception handling in Java, which is dependent on methods, since returns may occur inside the command body. In this case, elements concerning method definition and method return are tangled in the semantic equation for exception handling statement. Furthermore, method definition scatters the denotational specification, since its influence in other contexts must be described *in loco*.

General language properties comprehend language features which may affect the semantics of several constructs. This category includes dynamic type checking in strong-typed languages, and error handling. The impact of this sort of information context in the modularity of denotational specifications may be expressed by: (i) the influence of those properties are tangled in the semantic equations defining affected constructs; (ii) the specification of those properties may are scattered through the semantic equations defining affected constructs. It is also important to highlight that there is not a single section – or module – in the specification which addresses each property. In other words, the definition of such properties cannot be modularized since they crosscut several linguistic constructs.

This work focuses on the unsatisfactory degree of modularity and extensibility [Meyer 1997] which can be achieved in denotational semantics definitions of large scale programming languages, due to the lack of adequate mechanisms for the propagation of context information. As described in Section 3, inclosing structure context and enforcement of language properties are not yet addressed in current models for denotational semantics definition. As a consequence, incremental definition of large scale program-

ming languages is a complex and error-prone process, because including new constructs into an existing specification may require that modules defining other language feature be restructured. Another problem associated with these difficulties is the fact that one of the main purposes of formal semantics, the unambiguous communication of language constructs meanings, may not be achieved, since it may not be possible to understand the semantics of a construct in isolation. Yet, another problem is that referential transparency is not always guaranteed, since equations may depend on data stored in global area, such as the environment.

Additionally, denotational semantics definitions are not extensible, because small changes to language may produce widespread changes through the whole definition; for instance, incorporating sequencers in the specification of a language using direct semantics requires that all semantic equations be rewritten to accommodate continuations. These problems have direct impact on semantic definition scalability, since the number of details to be addressed is not proportional to the number of constructs of the language.

On the other hand, crucial information about the language may be obscured by several details in the semantic equations, which make it hard to fully understand the characteristics of some constructs. This leads to an ever higher complexity of using formal semantics specifications in the proof of language properties, since interleaved elements in the equations cannot always be abstracted away.

3. Context Handling in Current Approaches

Several approaches addresses the modularity of denotational semantics, among which outstand action semantics [Mosses 1992], monadic semantics [Liang et al. 1995, Wadler 1990], and monadic action semantics [Wansbrough and Hamer 1997].

Action semantics [Mosses 1992] allows writing readable denotational semantics definitions, since semantic equations resemble informal writing. However, this model does not adequately address the handling of all context information presented in Section 2, and for this reason, the definition of some features tends to be tangled in the semantic equations. Although there are elegant techniques for abstracting away environments and stores, there is no way of hiding destination and language properties contexts. As a matter of fact, although assignments and conditional statements are not directly dependent on continuations, in the action semantics definition of Pascal [Mosses and Watt 1993], equations for these constructs must consider the existence of continuations. Furthermore, context propagation of inclosing structure may not be explicitly defined and it is not possible to separately define dynamic type checking rules.

Monadic semantics [Liang et al. 1995, Moggi 1991, Wadler 1990] elegantly allows the encapsulation of stores, environments, and continuations, and additionally permits the modelling of non-determinism. However, this technique does not offer appropriate mechanisms to handle multiple destinations of constructs, and, as occurs with action semantics, there is no adequate treatment for inclosing structure and general properties enforcement.

In monadic action semantics [Wansbrough and Hamer 1997], actions are defined by means of monads, leading to the modelling of continuations in action semantics, without losing the readability of the model. However, this model shares the boundaries of action semantics, since it does not provide adequate context handling.

```
1  module Expressions
2      ignore [ \n\r\t]
3      token num = [0-9]+ is asInt
4      exp         ::= exp "+" term
5                    |  term : term
6      term:Exp    ::= term "*" factor
7                    |  factor : factor
8      factor:Exp  ::= "(" exp ")" : exp
9                    |  num
10     function e : Exp -> Int
11     e [exp1 "+" exp2] = e exp1 + e exp2
12     e [exp1 "*" exp2] = e exp1 * e exp2
13     e [int] = int
14 end
```

Figure 1. Example of construct definition in Notus.

The solutions found in literature only offer mechanisms to handle information propagated from construction antecedents, and solve in a partial way the problem of multiple destinations. In addition, inclosing structure context is not modularized, and its transmission is usually done by means of environments, which leads to high coupling and low modular cohesion. Furthermore, the enforcement of language properties is not modularized, and usually these features is scattered through the semantic equations. Therefore, modular propagation of context information remais an open problem in Programming Languages.

4. Outline of the Solution

The solution to the problem presented in Section 2 comprehends the definition of a new technique in which: contexts can be modularly defined and propagated; in an incremental definition, the influence of new constructs on existing equations can be modularly defined.

A domain specific functional language, named Notus [Tirelo and Bigonha 2007], is currently under implementation, and allows: package and modular organization and visibility control; lexis, and concrete and abstract syntax definition; syntactic and semantic domains definition; semantic functions e equations definition. Expressions in Notus are inspired in Haskell. Languages in Notus may be defined in an incremental way, so that one may start with a small core language, which is successively extended by the inclusion of new modules. Each module may define related language constructs, by giving its syntax and semantics, and may define how to affect existing modules, as explained in Section 6. Contexts in Notus are special semantic domains, and are defined in Section 5.

Figure 1 shows an example a core language composed by expressions. The lexis of the language ignores white spaces (line 2), recognizes integer numbers (line 3), and the operators and parenthesis used in the grammar (lines 4-9). The syntax domain of token `num` is `Int`, which is deduced from the domain of the built-in function `asInt`. The concrete grammar (lines 4-9) defines arithmetic operations with numbers, addition, multiplication, and parenthesized expressions. The syntactic domains are implied from the grammar by the following rules: since no domain is declared to `exp`, it is considered to

```
1 module Functions
2     import Expressions
3     token id = [a-zA-Z]+
4     extend factor ::= id "(" exp +- "," ")" : [id "(" exp+ ")"]
5     function apply : Id -> Exp* -> Int
6     ... // definition of function apply
7     e [id "(" exp+ ")"] = apply id exp+
8 end
```

Figure 2. Example of language extension in Notus.

be in domain `Exp`; term and factor are in domain `Exp`. The rules of the abstract grammar, are composed from the rules in the concrete grammar, by replacing each variable by its domain; some rules, such as the rules in lines 5, 7, and 8, must be ignored in the abstract grammar since they are used to define precedence. Line 10 defines the semantic function, `e`, which maps expressions into integer numbers. Lines 11-13 define the semantic equations for each rule in the abstract grammar.

Figure 2 defines an extension of the language in Figure 1. Line 3 include a new token, `id`, in the lexis. Line 4 defines an extension of the grammar by adding a new rule having `factor` as its left-hand side; this new rule defines function calls, which are an identifier followed by list of expressions separated by commas, and enclosed in parenthesis. The abstract grammar ignores the separators and considers only the identifier and the list of expressions; parenthesis is kept in the definition for readability purposes. Line 7 defines the semantic equation for function call, using the auxiliary function `apply` to switch the called function.

5. Context-aware Denotational Semantics

Context abstraction and propagation in Notus allows information to be transmitted through constructs without polluting their semantic equations. Some language constructs, such as literal values, are independent of the context, which, for this reason, must be transparent in their definition; some constructs, such as conditional statements, just propagate context information, even without handling them directly.

5.1. Context Domains

In Notus, contexts are semantic domains, and are defined as tuples composed by labeled context information. A context definition has form:

$$\mathtt{T} = \textbf{context}(\mathtt{label}_1 : \mathtt{domain}_1, \mathtt{label}_2 : \mathtt{domain}_2, \cdots, \mathtt{label}_n : \mathtt{domain}_n)$$

where T is a domain name, and each $\mathtt{label}_i$ labels context information, whose values are in domain $\mathtt{domain}_i$.

Context information may be classified with respect to its use and propagation. For input and output sequences and stores, each update cancels the previously associated value, so that only the new associated value is necessary; such context information is classified as *ephemeral*. On the other hand, values associated with environments and structural context may be used after updates associate new values; for this reason they

are classified as *persistent*. Optionally, each label in a context domain definition may be marked with its class, **ephemeral** or **persistent**; context information with no class mark is considered to be persistent. Let L be a language whose context is composed by stores and environments, as shown in the following definition:

$$\texttt{T} = \textbf{context}(\textbf{ephemeral}\ \texttt{store} : \texttt{S}, \texttt{env} : \texttt{R})$$

where R e S represent, respectively, domains of stores and environments. In this example, store is ephemeral and environment is persistent.

5.2. Context Expression and Pattern

Context information may be handled by means of the expressions: context constructor, context selector, context update, and context occlusion. In addition, the context pattern may be used in pattern matching expressions.

Context Constructor. The context constructor expression defines a value in a context domain with given initial values, and has the form $\texttt{T}(\texttt{label}_1 = \texttt{exp}_1, \texttt{label}_2 = \texttt{exp}_2, \cdots, \texttt{label}_n = \texttt{exp}_n)$, where `T` is a context domain, each $\texttt{label}_i$ is a context label, and $\texttt{exp}_i$ is an expression whose value is in the domain associated with $\texttt{label}_i$. Each label of `T` must appear at most once in such expression, and they may appear in any order; non-initialized context information is considered undefined. For instance, if `T` is the context domain defined in Section 5.1, then the expression `T(store = \loc->unused, env = \id->unbound)` defines a context, whose store maps all locations to `unused`, and whose environment maps all identifiers to `unbound`.

Context Selector. The context selector expression produces the value associated with some label in a context value, and has form `label t`, where `t` represents a value in a context domain `T`, whose labels set contains `label`. If the information labeled with `label` is undefined, then an execution error is issued. For example, let `t` be in domain `T` defined in Section 5.1. The value of expression `store t` is the value associated with `store` in `t`.

Context Update. The context update expression creates a new context value by updating some information in an existing context value, and has form

$$\texttt{t}[\texttt{label}_1 \leftarrow \texttt{exp}_1, \texttt{label}_2 \leftarrow \texttt{exp}_2, \cdots, \texttt{label}_n \leftarrow \texttt{exp}_n]$$

where `t` is a value in a context domain `T`, each $\texttt{label}_i$ is a label in the label set of `T`, and each $\texttt{exp}_i$ is an expression whose value is in the domain associated with $\texttt{label}_i$. For instance, let `t` be a value in domain `T`, defined in Section 5.1, and `s` be a value in domain `S`. The value expression `t[store <- s]` is a new context whose store is `s` and whose environment is the environment of `t`.

Context Occlusion. The context occlusion expression operates on two contexts, t_1 and t_2, and its purpose is to create a new context t from t_1 whose ephemeral information are updated with values from t_2. This expression has form $\texttt{t}_1 * \texttt{t}_2$, where $\texttt{t}_1$ and $\texttt{t}_2$ are contexts in domain `T`, and the result is a context `t`, where for each label `label` in `T`, the expression `label t` evaluates to:

- undefined, if `label` $\texttt{t}_1$ and `label` $\texttt{t}_2$ are undefined;

```
1 V = ... // domain of values
2 Io = context(ephemeral i: V*, ephemeral o: V*|{error})
3 function f : Io -> Io
4 f io[() <- i] = io[o <- error]
5 f io[v:v* <- i, v1* <- o] = io[i <- v*, o <- v1* ++ (v)]
```

Figure 3. Example of context pattern.

- label t_1, if label t_2 is undefined, or label is ephemeral;
- label t_2, otherwise.

For example, let $t_1 = T(\text{store} = s_1, \text{env} = r_1)$ and $t_1 = T(\text{store} = s_2, \text{env} = r_2)$ be contexts in domain T defined in Section 5.1. The value of expression $t_1 * t_2$ is the context $t = T(\text{store} = s_1, \text{env} = r_2)$

Context Pattern. The context pattern is used in pattern matching expressions, and has the form $\text{patt}[\text{patt}_1 \leftarrow \text{label}_1, \text{patt}_2 \leftarrow \text{label}_2, \cdots, \text{patt}_n \leftarrow \text{label}_n]$, where patt is a domain pattern or a context pattern, each label_i represents a label, and each patt_i is a valid pattern for the domain of label_i. Let:

- $T =$ **context**$(\text{label}_1 : \text{domain}_1, \text{label}_2 : \text{domain}_2, \cdots, \text{label}_n : \text{domain}_n)$ be a context domain;
- $t_0 = T(\text{label}'_1 = \text{value}_1, \text{label}'_2 = \text{value}_2, \cdots, \text{label}'_m = \text{value}_m)$ be a value in domain T;
- $p = t[\text{patt}_1 \leftarrow \text{label}''_1, \text{patt}_2 \leftarrow \text{label}''_2, \cdots, \text{patt}_k \leftarrow \text{label}''_k]$ be a pattern applicable to T.

Pattern p matches value t_0 if for each label''_i in p, there is a label'_j in t_0, such that $\text{label}''_i = \text{label}'_j$ and patt_i matches value_j. In other words, the pattern associated with each label in pattern p must match the corresponding value in t_0.

For instance, consider the fragment of code in Figure 3, which defines a domain V, a context domain Io, representing input and output context, and a function f in Io $\rightarrow$ Io, which echoes in the output sequence the first value in the input sequence. Context Io is composed by ephemeral information i, representing lists of values, and o, representing either lists of values or the constant error. The first clause of function f (line 4) defines that if the input sequence of a context value io is the empty list, then the result is the context created from io where the output information is error. The second clause (line 5) defines that if the input sequence is a list having at least one element, and the output sequence is a list of values, then the resulting context appends the first value of the input in the end of the output.

5.3. Context Expansion

Incremental definitions usually demands the context structure be redefined. For instance, when sequencers are added to a language, it becomes necessary to include inclosing structure information (see Section 5.4) in the context domain tuple.

It may be done in Notus by means of the expand clause, whose goal is to add context information to a given context domain, has the form:

expand T **with** $\text{label}_1 : \text{domain}_1, \text{label}_2 : \text{domain}_2, \cdots, \text{label}_n : \text{domain}_n$

where each $\texttt{domain}_i$ is the domain of some new information labeled with $\texttt{label}_i$. Optionally, each label may be marked with **ephemeral** or **persistent**.

For instance, let $\texttt{T} =$ **context**(**ephemeral** $\texttt{store} : \texttt{S}, \texttt{env} : \texttt{R}$) be a context domain. The following clause adds input and output information, both of them ephemeral and represented by lists of values in domain V:

expand T **with ephemeral** $\texttt{input} : \texttt{V}\star$, **ephemeral** $\texttt{output} : \texttt{V}\star$

Each context label may appear only once in context domain definitions and expansions. If some label is redefined, an compilation error is issued.

5.4. Structural Context

Inclosing struture context, or just structural context, of a language construct is constituted by its ancestral nodes in the abstract syntax tree of the program. Each ancestral node is associated with information present during the evaluation of that node. For instance, consider the *while* and *break* statements in the following C code fragment:

```
while (condition1) { C1; if (condition2) break; C2 } C3
```

The structural context of the break statement contains the AST node representing the inclosing while statement. In addition, that node may be associated with the destination of the while, represented by statement C_3, which is necessary for interrupting the repetition.

In Notus, context domains have a persistent context information labeled with structure, which stores information related to structural context. Values associated with this label are in domain $\texttt{Far}^*$ and consist of stacks of function application records, which are elements from domain Far. Each record stores a function name and the arguments of the application.

For manipulating structural contexts, each context domain T is associated with a function $\texttt{push} : \texttt{T} \rightarrow \texttt{Far} \rightarrow \texttt{T}$, where push t *far* is a new context t' built from t, in which *far* is push onto the stack structure of t; function push is defined by:

$$\texttt{push}\ t\ \textit{far} = t[\texttt{structure} \leftarrow \textit{far} : (\texttt{structure}\ t)].$$

Function push is implicitly called in every function application whose arguments contain a context value, so that for any application $f\ \alpha_1\ \cdots\ \alpha_n$, the compiler replaces each context argument, t, by $t' = \texttt{push}\ t\ \textit{far}$, where *far* is the function application record corresponding to the modified application of f.

Example. Let V be a domain of values, T be the domain of contexts, f, g, and h be functions, a be an element in domain V, t be an element in domain T. Considering function f is defined by $f\ v\ t = g\ v + h\ v\ t$, a function application $f\ a\ t$ is transformed by the compiler into the application $f\ a\ t'$, where $t' = \texttt{push}\ t\ \textit{far}$, and *far* represents the application $f\ a\ t'$.

Context information stored in the stack of function application records may be accessed by means of the *match* expression, which has the form **match** p **with** e **in** e', where p is a structural context pattern, and e and e' are expressions. The value of this match expression is the value of e' if the value of e is a context value t and pattern p matches at least one function application record in the stack labeled by structure in t.

If pattern p matches no record in the stack, then an execution error is issued; if it matches more than one record, only the topmost matched record is considered. Identifiers defined in p are bound to the values in the matched record, and may be used in expression e'.

Example. Let $f : \texttt{Int} \rightarrow \texttt{T} \rightarrow \texttt{Int}$, $g : \texttt{Int} \rightarrow \texttt{T} \rightarrow \texttt{Int}$ be functions defined by:

$$\begin{array}{lcl} f\ a\ t & = & g\ a\ t \\ g\ a\ t & = & 1 + g\ (a-1)\ t \\ g\ 0\ t & = & \textbf{match}\ f\ b\ \textbf{with}\ t\ \textbf{in}\ b \end{array}$$

The value of function application $f\ a\ t$ is given by the application $g\ a\ t$; the compiler defines that a function application record corresponding to this call is pushed onto the stack of t. Then, a series of recursive applications of function g is executed, and once the base of the recursion is reached, it recovers and produces as result the argument in the application of f.

Structural context patterns in Notus may match single function application records, or may be composed using the direct origin operator **at**.

The *function application record pattern* (far pattern, for short), has the form $f\ p_1\ p_2\ \cdots\ p_n$, where f is a function name, and each p_i represent a pattern which does not represent a far pattern. For instance, consider the definition of $g\ 0\ t$ in the previous example; pattern $f\ b$ matches the most recent application of function f, provided it is found in the record stack. The *direct origin pattern* has the form p_1 **at** p_2, where p_1 e p_2 are far patterns. This pattern matches consecutive records, $\texttt{far}_1$ and $\texttt{far}_2$, in the structural context stack, such that p_1 matches $\texttt{far}_1$, p_2 matches $\texttt{far}_2$, and $\texttt{far}_1$ immediately follows $\texttt{far}_2$ in the stack. For instance, the expression **match** $f\ a$ **at** $h\ b$ **with** t **in** $a + b$ matches applications of function f directly actived from the application of function h; the argument of f is bound to a, and the argument of h is bound to b, which are used in the result.

5.5. Example of Semantics Using Structural Context

Consider the fragment specification of the C programming language of Figure 4, in which `E` is the syntactic domain of expressions, `C` is the syntactic domain of commands, `T` is the domain of contexts, $\texttt{K} = \texttt{T} \rightarrow \texttt{T}$ is the domain of continuations, `de` is the semantic function for expressions, and `dc` is the semantic function for commands. Auxiliary function `getW` (lines 11-13) takes a context t and matches an application of function `dc` immediately following an application of function `dc` to an AST node representing a while command. The destinations of the while and its body, `k1` and `k2`, along with the while context, `t'`, are returned as a tuple in domain `W` (defined in line 10).

The semantic equations for the while command (lines 15-17) defines its behavior without considering the existence of sequencers. In line 18, the denotation of the break command in context `t` captures the destination of the inclosing while, `k'`, and its context, `t'`, from `t`, and ignores the current continuation, `k`, by passing to `k'` the context `t' * t`; this context is the occlusion of context `t'` with `t`, so that ephemeral information, such as the store, come from the context of the break command, and persistent information, such as the environment, come from the context of the while. The denotation of continue command is very similar to the denotation of break, having as destination the execution of the while.

```
1  module Commands
2      import Expressions
3      c ::= "while" "(" e ")" c : ["while" e c]
4          |  "break" | "continue"
5          |  ... // other commands
6      function dc : C -> K -> K
7      W = W(K,K,T)
8      function getW : T -> W
9      getW t = match dc c k2 at dc ["while" e c] k1 t'
10                 with t in W(k1,k2,t')
11     dc ["while" e c] k =
12         de e; \v -> if isZero v then k
13                     else dc c (dc ["while" e c] k)
14     dc ["break"] k t = let W(k',_,t') = getW t in k' (t' * t)
15     dc ["continue"] k t = let W(_,k',t') = getW t in k' (t' * t)
16 end
```

Figure 4. Fragment of Formal Specification of Commands of the C Language.

6. Module Tranformations

6.1. Transformation Definition

A module transformation function, or module transformer, creates an abstract syntax tree (AST) from a module AST, by modifying some of its definitions. A simple example consists of adding continuations in a specification. Consider that, in an incremental definition, a set of modules, $M_1, M_2, \cdots, M_k$, defines direct semantics of constructs, and that module M_{k+1} defines sequencers, which demand the use of continuations. To avoid rewriting the existing modules, it is sufficient to apply a module transformer which alters the signature of semantic functions and the semantic equations to incorporate continuations; it is also possible to transform semantic domains. The examples throughout this section refer to this example of adding continuations to an existing definition.

Transformers are ordinary functions which operate on predefined domains representing Notus constructs. A comprehensive list of modules, domains, and functions are available in [Tirelo and Bigonha 2007]. A module transformer has domain `Module` $\rightarrow$ `Module`, where `Module` is a predefined domain representing modules in Notus. It means that the effect of such transformation is to create a new module from a given module, by individually processing its components. Transformers may be defined and applied in any module in the specification, provided it obeys the rules set in this section.

6.2. Transformer Application

Applications of module transformer are specified in the import lists of modules. For instance, if module M_1 imports module M_2, applying transformer f, then the import list of module M_1 contains $f\ M_2$. Function f may be defined in module M_1 or in any other module it imports. In addition, transformer f is also applied to the modules imported by M_2, to the modules they import, and so ever, which keeps the programmer from listing all imported modules.

```
module Ma
    function f : A -> B;
    f a = ...
    ...
end

module Mb
    function g : X -> Y;
    g x = ...
    ...
end
```

```
module Mc
    import t Ma, ...;
    function t : Module -> Module;
    ... usage of (f a) ...
end

module Md
    import Mc, t Mb, ...;
    ... usage of (g x) ...
end
```

Figure 5. Example of Modules of a Specification.

In the example of Figure 5, module `Ma` defines function `f`, and module `Mb` defines function `g`; module `Mc` defines transformer function `t`, which is applied to module `Ma`; module `Md` applies transfomer t, imported from module `Mc`, to its imported module `Mb`.

6.3. Transformer Graph

Transformer applications in a Notus specification are modeled by a transformer graph, which is a directed and labeled graph $G = (V, E)$, such that V is the set of modules in the specification, and the edge set, E, is defined by:

$$E = \{(m_1, m_2) \in V \times V \mid \text{module } m_1 \text{ imports module } m_2\}.$$

Each edge in E is labeled with the transformer function used in the import, or by the identity function, if no transformer is applied in the import.

Given a path $\pi = \langle e_1, e_2, \cdots, e_n \rangle \in E^*$ in G, the *transformer of* π is the function $f_\pi = f_1 \circ f_2 \circ \cdots \circ f_n$, where each f_i is the label of edge e_i. For instance, the transformer of the path from `Md` to `Ma` of the graph corresponding to the modules in Figure 5 is $i \circ \mathtt{t}$.

A specification is *consistent* if for all pair of modules m_1, m_2 such that the elements of m_1 are used by m_2, the elements of m_1 are uniquely interpreted in m_2. In the transformer graph, for each pair of vertices m_1, m_2, such that there are more than one path from m_1 to m_2, $f_\pi = f_{\pi'}$, for all π, π' from m_1 to m_2. By this definition the specification composed by the modules in Figure 5 is consistent. If a specification is not consistent, then a compilation error is issued. In a consistent specification, each module m considers a imported module m' is the module $f_\pi\ m'$, where π is a path from m to m'. In the example of Figure 5, module Md uses module $(i \circ \mathtt{t})\ \mathtt{Ma}$.

7. Conclusion

A new solution to the scalability of denotational semantics specifications of programming languages is proposed in this paper. This solution, based on oblivious context transmission and module transformations, improves the readability of semantic equations by adequately handling interfering language constructs, such as repetition statements and sequencers. It permits changes to existing equations in order to accommodate context information transmission, which is required by new constructs in an incremental definition,

using only features with which language designers are familiar. For instance, a transformation from direct to continuations semantics may be achieved by including a single module in the specification.

The main advantage of the proposed method is that the context modularization problem, defined in Section 2, is solved in a simple and independent manner. Antecedents are abstracted away an argument in a context domain, and are only revealed when necessary. The intended use of each context element is made explicit by means of the classification of information as ephemeral or persistent. The inclusion of new context information is done in a uniform way and does not require adapting stores and environments.

Existing approaches for modular denotational semantics also provide techniques to make antecedents transparent, but at a higher cost. Adapting the kernel of action semantics solution consists of changing a monolithic kernel, whose difficulty is highlighted by [Wansbrough and Hamer 1997]. In monadic semantics, those changes may require the definition of new monad transformers, which not always compose in an elegant way, requiring the definition several new "lift" functions. In monadic action semantics, the kernel of action semantics is modularized, but its change suffers from the same difficulties of monadic semantics.

The modularization of structural context of constructs is a problem not solved by current approaches. The presented solution is based on properties of context domains. Once a function having a context argument is applied, its context must be stored to be used when needed in inner constructs. Multiple destinations in constructs, when caused by the inclosing structure of a construct, are also modularized by this solution. However, the presented approach does not yet address the problem of modularizing multiple destinations of labeled *goto* statements of Pascal and C.

Language properties enforcement usually causes tangled code in semantic equations. In the proposed approach, this problem is solved by defining a single transformer, which directs this inclusion to be transparently done by the compiler, this enforcement may be confined to a single module.

The current stage of this work includes the formalization of Notus, its implementation, and the solution of the multiple destination problem caused by *goto* statements. Future work comprehends the definition of new standards for denotational semantics and further studies of its properties and consequences.

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.

References

Bigonha, R. S. (2006). The Scalability Problem of Denotational Semantics. Talk presented in the Brazilian Symposium on Programming Languages 2006.

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.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning.

Liang, S., Hudak, P., and Jones, M. (1995). 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. ACM Press.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall.

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. (1979). SIS, Semantics Implementation System: Reference manual and user guide. DAIMI MD30 MD–30, Dept. of Computer Science, Univ. of Aarhus.

Mosses, P. D. (1992). *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Mosses, P. D. and Watt, D. A. (1993). Pascal action semantics, version 0.6. `http://www.brics.dk/~pdm/papers/MossesWatt-Pascal-AS/`.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058.

Scott, D. and Strachey, C. (1971). Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab.

Tirelo, F. (2005). Semântica Multidimensional de Linguagens de Programação – Proposta de Tese de Doutorado. Technical report, Laboratório de Linguagens de Programação, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais.

Tirelo, F. and Bigonha, R. S. (2007). The Notus Language. Technical report, Laboratório de Linguagens de Programação, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil.

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.

Wadsworth, C. (1978). Mathematical Background to Denotational Semantics. Lecture notes on denotational semantics course at UCLA.

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.