Separation of Concerns in Denotational Semantics Descriptions

Roberto S. Bigonha
Universidade Federal de Minas Gerais
bigonha@dcc.ufmg.br

Fabio Tirelo
Google Inc
ftirelo@google.com

Guilherme H.S. Santos
Universidade Federal de Minas Gerais
guisousa@dcc.ufmg.br

ABSTRACT

Denotational semantics is a powerful and elegant formalism for describing the meaning of programming language constructs, but it is used less than it should. Apparently, the difficulty of reading formal semantic definitions is inherent to the way they are organized. This paper presents a semantics definition style based on the concept of components in order to provide legibility to descriptions in this formalism. The idea is to remove context dependence from the semantic equations. Consequently, the equations in this style only specify the direct mapping from language constructs to their denotations by means of denotational components in an easy and readable way, and the details, such as context handling, are encapsulated away.

Keywords

Denotational Semantics, Legibility, Modularity, Semantic Components

1. THE ROOTS OF ILLEGIBILITY

In the industry, programming languages are described by means of a formal presentation of their syntax based on context free grammars together with an informal description of their semantics. Even when a formal definition of the language is publicly available, it is rarely read by programmers and computer scientists.

According to P. Mosses[7], one of the reasons for this limited use of formal semantics in the industry is the difficulty most programmers and computer scientists have in dealing with the mathematical apparatus of formal definitions.

If Backus-Naur form has been established as a universal notation for defining programming languages syntax, formal semantics methods have never achieved similar success. Probably this is due to the fact that no formal semantics method has the simplicity of the syntactic formalisms, and also that, in denotational semantics, although the semantics of a language construct should depend only on the semantics of its immediate constituents, there are always, in the semantic equations, explicit dependences on other elements, such as the construct's context.

Tirelo et alii[11] have identified that the context in which the meaning of a construct is defined can split in: (i) *antecedents* (ii) *destination* (iii) *locality* as shown in Fig. 1.

The *antecedents* of a construct's context comprises the effects of what has been executed priorly in the program. In general these effects are propagated to the construct by entities like store and enviroment. For example, the values previously assigned to variables are part of the context in which an expression is evaluated.

The *destination* of a construct provides the context to which the effects of its execution are to be sent. This is usually modelled by the notion of continuations. For example, in a statement sequence, the destination of the results produced by the execution of the first statement are the commands that follow it.

And the *locality* is the context given by the construct's enclosing structure in the abstract syntax tree. The semantics of the C `break` statement, for instance, depends on whether it occurs inside a loop or in a `switch` statement.

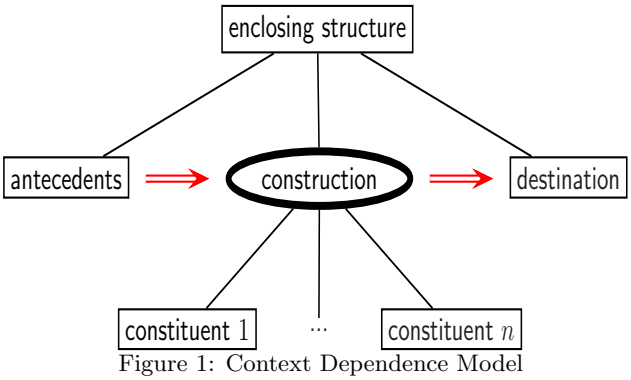

Figure 1: Context Dependence Model

In order to cope with all these facets of the context, the semantic equation of a construct must be provided with apropriate parameters, in addition to those that specify its constituents. The semantics specification requires that these context parameters be transmitted to the denotations of the construct's constituents and also to its destination. The need to explicitly deal with context produces an undesirable dependence relation among equations and the related domain apparatus in which they are defined, impairing their legibility.

Moreover, the abstraction mechanisms of λ-calculus, considered insufficient to properly encapsulate definition details

SAC'14 March 24-28, 2014, Gyeongju, Korea.

of semantic domains, aggravate even more the legibility issue [3, 6, 7, 12].

In summary, context dependence, the lack of appropriate modularization mechanisms and the need to always provide complete semantics definitions make them very intricate, and, thus, the legibility of formal descriptions of real size programming languages like C++ and Java become completely undermined.

The purpose of the work is to provide a method for organizing denotational semantics descriptions in order to enhance legibility. The method is based on removing the context dependence from all semantic equations and constructing a separate module of denotational components to encapsulate context details. This style of organizing definitions rescues the idea that the semantics of a construct only depends on that of its immediate constituents.

2. THE CLASSICAL STYLE

The formal description of the toy programming language Small, defined by M. Gordon [2], will be used to demonstrate the use of components and to highlight the improvement in legibility that can be achieved. The abstract syntax of Small is presented in Fig. 2. The equations and definitions in Fig. 3, 4 e 5 are an adaptation from Gordon's original definition.

```
Primitive Syntactic Domains:
Ide    =    domain of identifiers I
Bas    =    domain of basic constants B
Opr    =    domain of binary operations O
Bool   =    {true,false}

Compound Syntactic Domains:
Pro    =    domain of programs P
Exp    =    domain of expressions E
Com    =    domain of commands C
Dec    =    domain of declarations D

Syntactic Clauses:
Pro    →    program C
D      →    const I = E | var I = E | proc I(I1);C
       |    fun I(I1);E | D1;D2
C      →    E1 := E2 |output E
       |    if E then C1 else C2 | while E do C
       |    begin D;C end | C1;C2
E      →    B | true | false | read | I | E1 O E2
       |    E1(E2) | if E then E1 else E2
```

Figure 2: Abstract Syntax of Small

The domains defined in Fig. 3 intend to model imperative language whose expressions may have collateral effects and several types of errors may be flagged during program execution. Procedure and functions are limited to have just one parameter to simplify the presentation. Likewise, the definitions of binary operation (O and of integer constants (B) are left unspecified.

Important domains are those modelling expression, command and declaration continuations (Ec, Cc, Dc), machine states (Store) and environments (Env). The continuation domains model the destination of the effects of executing a construct, the state represents an abstract structure to store values and the environment defines the meaning of names in the program. Environment and machine state model the antecedents of a construct.

```
Primitive Semantic Domains:
 Num = {0, 1, 2, ··· }          - numbers n
 Boolean = {true,false}         - booleans b
 Loc = Num                      - locations l
 Bv = Num + Bool                - basic values e
 Id = domain of identifier      - identifiers I

Compound Semantic Domains:
 Ans = {error,stop}+Rv×Ans      - answers a
 Rv = Bool + Bv                 - r-values v
 File = Rv*                     - files i
 Dv = Loc + Rv + Proc + Fun     - denotable d
 Ev = Dv                        - expressible e
 Sv = Rv + File                 - storable x
 Env = Id →[Dv+{unbound}]       - environments r
 Store = Loc→[Sv+{unused}]      - stores s
 Dc = Env→Store→Ans             - continuation u
 Ec = Ev→Store→Ans              - continuation k
 Cc = Store→Ans                 - continuation c
 Fun = Ec→Ev→Store→Ans          - fun values f
 Proc = Cc→Ev→Store→Ans         - proc values p

Denotation Domains:
 Pd = File → Ans                - programs
 Dd = Env→ Dc→ Store→ Ans       - declarations
 Ed = Env→ Ec→ Store→ Ans       - expressions
 Cd = Env→ Cc→ Store→ Ans       - commands

Semantic Functions:
 P:Pro → Pd                     - programs
 D:Dec → Dd                     - declarations
 R:Exp → Ed                     - expressions
 E:Exp → Ed                     - expressions
 C :Com → Cd                    - commands
```

Figure 3: Semantic Domains (adapted from [2])

Domain Dv is the domain of denotable values, with which Small identifiers can be associated in the environment. To keep the formulation simple, identifiers used in Small programs and their respective denotations are considered the same, i.e, I denotes elements of Ide and of Id. The context of its use should be sufficient to resolve this overloading.

Domain Sv is that of storable values which can be associated with locations in the machine states. The values passed to expression continuations belong to the domain Ev of expressible values. The basic values Bv represent booleans and numbers. The domain of program denotations Pd represents functions that model Small programs as a whole. Functions in this domain must deal with the program's input file, its initial context and produce the final answer of its execution.

The propagation of the semantic effects of Small constructions are modeled by the domains Dc, Ec and Cc of declaration, expression and command continuations, respectively.

Expression denotations are elements of domain Ed, command denotations are in Cd, and Dd is the domain of the declaration denotations.

The auxiliary functions in Fig. 4 implement a set of fundamental operations over members of semantic domains. Most of these functions follow the continuation style in order to be adhrent to the adopted model.

```
• apply:Opr×Rv×Rv → Ec → Store → Ans
  apply(o,v1,v2) k s = k (v1 o v2) s
• bool:Boo → Bool
  bool = λb.b = true → true, false
• Bool?:Ec → Ev → Store → Ans
  Bool?  = λk e s.isBool e → k e s, error
• cond:D × D → Bool → D
  cond = λ(d1,d2) b.b → d1, d2
• contents:Ec → Ev → Store → Ans
  contents = λk e s.isLoc e →
          (s e = unused→error, k(s e)s) , error
• deref:  Ec → Ev → Store → Ans
  deref = λk e s.isLoc e→contents k e s , k e s
• fix:[D → D] → D
  fix = λf.compute the fixpoint of f
• hd:D* → D
  hd = λ(d1,d2,···,dn).d1
• Loc?:Ec → Ev → Store → Ans
  Loc?  = λk e s.isLoc e → k e s, error
• new = Store → [Loc + {error}]
  new = λs.∃ free location l in s→ l, error
• number:Bas → Num
  number = λn.convert n to Num
• null:D* → Bool
  null = λd*.d*=() → true, false
• ref:Ec → Ev → Store → Ans
  ref = λk e s.new s = error → error,
                update (new s)(k(new s)) e s
• tl:D* → D*
  tl = λ(d1,d2,···,dn).(d2,···,dn)
• update:Loc → Cc → Ev → Store → Ans
  update = λl c e s.isSv e → c(s[e/l]),error
```

Figure 4: Auxiliary Functions (adapted from [2])

3. THE UNDERLYING IDEA

The proposed style of organizing denotational descriptions is inspired in Peter Mosses' concepts of components [8, 9, 10] that he has used to improve reusability of action semantics and structured operational semantics descriptions.

The proposed method basic idea is illustrated with the description of the **if-then-else** statement of Small, which is defined by the folowing equation extracted from Fig. 5:

$$\begin{aligned}&\mathcal{C}[\![\texttt{if E then C}_1\texttt{ else C}_2]\!]\texttt{r c s} =\\ &\quad\mathcal{R}[\![\texttt{E}]\!]\ \texttt{r}\ (\lambda\texttt{v s.Bool?}(\lambda\texttt{v s.}\\ &\qquad\texttt{cond}(\mathcal{C}[\![\texttt{C}_1]\!]\ \texttt{r c s},\ \mathcal{C}[\![\texttt{C}_2]\!]\ \texttt{r c s})\texttt{v})\texttt{v s})\texttt{s}\end{aligned}\tag{1}$$

This construct's context is defined by the enviroment $\texttt{r} \in \texttt{Env}$, the command continuations $\texttt{c} \in \texttt{Cc}$ and by the machine state $\texttt{s} \in \texttt{Store}$.

The use of this context is fundamental to convey the desired meaning to the command, but it certainly impairs readability. Note that the context, represented by r, c and s is mentioned 18 times in the equation (1), and it would be nice to be able to cross it out.

This can be achieved by means of a Turner-like combinator[1], which produces equation (2):

$$\begin{aligned}&\mathcal{C}[\![\texttt{if E then C}_1\texttt{ else C}_2]\!]\texttt{r c s} =\\ &\qquad\texttt{choice}(\mathcal{E}[\![\texttt{E}]\!],\mathcal{C}[\![\texttt{C}_1]\!],\mathcal{C}[\![\texttt{C}_2]\!])\ \texttt{r c s}\end{aligned}\tag{2}$$

```
..... ..... ..... Programs ..... ..... .....
P[[program C]] i = C[[C]] r0 c0 s0
   where r0 = λI.unbound
         c0 = λs.stop
         s0 = λl.unused)[i/input]
   N.B.:  input∈ Loc is a reserved location
.................Declarations.................
D[[const I = E]]r u s = R[[E]]r(λv s.u[v/I]s)s
D[[var I = E]]r u s =
    R[[E]] r(λv s.ref (λl s.u [l/I]s)v s)s
D[[proc I(I1;C)]]r u s =
    u((λc e s.C[[C]]r[e/I1]c s)/I)s
D[[fun I(I1;E)]]r u s =
    u((λk e s.E[[E]]r[e/I1]k s)/I)s
D[[D1 ; D2]]r u s =
    D[[D1]]r(λr1 s.D[[D2]]r[r1] (λr2 s.u(r1[r2])s)s)s
.................Expressions .................
E[[true]]r k s = k true s
E[[false]]r k s = k false s
E[[B]]r k s = k number(B) s
E[[I]]r k s = (r I = unbound) → error, k (r I) s
E[[read]]r k s = null (s input) → error,
       k (hd (s input)) s[tl (s input)/input]
E[[E1 O E2]]r k s =
    R[[E1]] r(λv1s.R[[E2]] r(λv2s.apply(O,v1,v2)k s)s)s
E[[E1(E2)]]r k s = E[[E1]]r(Fun?(λf s.E[[E2]]r(f k)s))s
E[[if E then E1 else E2]]r k s =
     R[[E]]r (λv s.Bool?(λv s.
         cond(E[[E1]]r k s,E[[E2]]r k s)v)v s)s
R[[E]] r k s =
    E[[E]] r (λe s.deref(λv s.Rv?  k e s)e s)s
..... ..... ..... Commands ..... ..... .....
C[[E1 := E2]]r c s = E[[E1]]r(Loc?λl s.R[[E2]]r
                        (λv s.update l c v s)s))s
C[[output E]]r c s = R[[E]] r(λe s.(e,c s))s
C[[if E then C1 else C2]]r c s =
     R[[E]] r (λv s.Bool?(λv s.
         cond(C[[C1]] r c s, C[[C2]]r c s)v)v s)s
C[[while E do C]]r c s = fix λf.R[[E]]r(Bool?
    (λv s.cond(C[[C]]r(λs.f r c s)s,c s)v))s
C[[E1(E2)]]r c s = E[[E1]]r(Proc?(λp s.E[[E2]]r(p c)s))s
C[[C1 ; C2]]r c s = C[[C1]] r(C[[C2]]r c)s
C[[begin D ; C end]]r c s =
      D[[D]]r (λr1 s.C[[C]]r[r1] c s) s
```

Figure 5: Classical Semantics of Small (adapted from [2])

where the combinator choice is defined as:

$$\begin{aligned}&\texttt{choice(E, C}_1\texttt{, C}_2\texttt{) r c s} =\\ &\qquad\texttt{E r (}\lambda\ \texttt{v s.Bool?(}\lambda\ \texttt{v s.}\\ &\qquad\quad\texttt{cond(C}_1\ \texttt{r c s, C}_2\ \texttt{r c s)v)v s)s}\end{aligned}\tag{3}$$

Note that the combinator parameters are solely denotations of the command's immediate constituents and the context. For clarity purpose, it is important to keep it that simple.

The next step is to abstract away the combinator definition from the description reader's eyes, placing it in a library of denotational components, and applying η reduction to equation (2) that becomes:

$$\mathcal{C}[\![\texttt{if E then C}_1\texttt{ else C}_2]\!] = \texttt{choice}(\mathcal{E}[\![\texttt{E}]\!],\mathcal{C}[\![\texttt{C}_1]\!],\mathcal{C}[\![\texttt{C}_2]\!])$$

This equation reveals that the semantics of Small **if-then-else** command is such that the value of E is to be evaluated

first, and if its value is **true**, the execution proceeds with command `C1`, otherwise command `C2` is to be executed.

Hopefully, to have this level of understanding of the meaning of the defined construction it is enough to know the `choice`'s interface, and there is no need to know details of the definition of this combinator.

To generalize this structuring process, consider the semantics h of a generic construct A defined in a context `z`:

$A \rightarrow r_0 B_1 r_1 B_2 \ldots B_n r_n$

$h : A \rightarrow Context \rightarrow Ans$

$h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!]\, z = g(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!], z)$

The functions h_i, for $0 \leq i \leq n$, give the semantics of A's constituents.

This semantics modelling follows the mapping structure displayed in Fig. 6, in which function g combines the semantics of the immediate constituents of `A` to produce its meaning. The function g does not show any dependency on the terminal symbols that occur on the right hand side of the production defining A. Only the nonterminals take part in the formulation. This means that each right hand side must imply in a new g, i.e., the dependence on terminal symbols are forged into the structure of g. This is so because although passing denotations of terminal symbols directly to g could hinder legibility, there are situations in which it may contribute to component reuse.

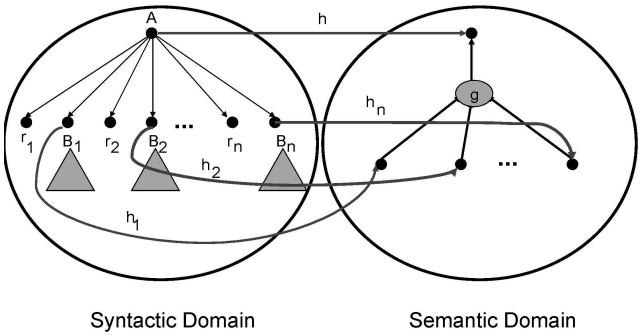

Figure 6: The Semantic Model

To encapsulate the flow of context information, consider the generic combinator K, defined as:

$$K\,(d_1, d_2, \ldots, d_n)\, z = g(d_1, d_2, \ldots, d_n, z)$$

Observe that K only operates over the denotations of the immediate constituents of A, preserving the meaning provided by g. Using K to rewrite the definition of h produces:

$$h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] z = K(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!])z$$

which may be simplified to:

$$h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] = K(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!])$$

It is recommended that each equation use just one combinator, avoiding enticing combinator compositions so as to rescue the central idea of the denotational semantics formalism that the meaning of a construct only depends on the meanings of its immediate constituents.

To achieve legibility, discipline and standardization are mandatory. So it seems reasonable to require that all combinators like K must have the standard type:

K: $D_1 \times D_2 \cdots \times D_n \rightarrow Context \rightarrow Ans$, for $n \geq 0$

where D_i, for $0 \leq i \leq n$, are domains of denotations or of special values associated with the production.

Typically, the parameters of a combinator should be only denotations. However, in order to favour reuse of components and yet preserving legibility, sometimes it is convenient to pass to the combinator especial values to determine some specific behavior, instead of writing several similar functions. The need for this arises when more than one production have the same nonterminals on their right hand sides, being distinguished only by the terminal symbols involved.

This process of encapsulating context should be applied to the semantic equations of all constructs in the language, producing a clean set of mapping, such as that of Fig. 10, which is much more legible than its counterpart in Fig. 5.

The claim is that to understand the component-based denotational semantics description of a given programming language all that is required is to know the interface of the used components, without any concern regarding details of their definitions.

Due to the continuous evolution of programming languages, it would be interesting to have a library of generic components that allows easy incorporation of new constructs to the languages. The more generic are the components the better will be the library, because the same components could be used to define many languages.

The challenge is to find a set of generic components capable of modelling the semantics of the most important constructs of popular languages, thus reducing the need to define new components whenever defining new programming languages. However, for space reasons, this problem is not addressed in this paper.

4. COMPONENT-BASED SEMANTICS

The information regarding context, i.e., environment, store and continuations, only appears on the definition of the main equation of the description, the definition of $\mathcal{P}[\![\texttt{Program C}]\!]$, which gives the meaning of Small programs. This is the moment the context should be properly and explicitly initialized, and from this point on, it flows implicitly throughout the description.

Thus, the equation for $\mathcal{P}[\![\texttt{Program C}]\!]$ in Fig. 5 should not be object of componentization and, without modification, it becomes part of the component-based description of Fig. 10

The componentization process consists in creating new components to replace the right hand side of the Small semantic equations. In case of Small declarations, these components are `cbinding`, `fbinding`, `pbinding`, `vbinding` e `elaboration`, which show how the denotations of the constituents of each declaration are to be combined to build the semantics of the respective construct. The resulting semantic equations from the componentization of Small declarations are presented in Fig. 7, and transported to Fig. 10.

In the componentization process, the type of the components must keep conformity to the structure defined in section 3: either the components parameters are denotations of constructs, e.g. `operation` and `pcall` or they are basic values, such as in `value`, `association` and `read` in Fig. 8.

The resulting component-based definition of Small is in Figura 10, which is the reference text for the semantics of the language Small.

The other equations, whose details can be abstracted by the reader, are presented in Fig. 2, 3, 4, 7, 8 and 9.

```
D⟦const I = E⟧r u s = cbinding(I,R⟦E⟧)r u s
where
   cbinding:Ide×Ed→Env→Dc→Store→Ans
   cbinding(I,d)r u s = d r(λv s.u[v/I]s)s
D⟦var I = E⟧r u s = vbinding(I,R⟦E⟧)r u s
where
   vbinding:Ide×Ed→Env→Dc→Store→Ans
   vbinding(I,d) r u s =
         d r(λv s.ref(λl s.u[l/I]s)v s)s
D⟦proc I(I1;C)⟧r u s = pbinding(I,I1,C⟦C⟧)r u s
where
   pbinding:Ide×Ide×Cd→Env→Cd→Store→Ans
   pbinding(I,I1,d) r u s =
        u((λc e s.C⟦C⟧r[e/I1]c s)/I)s
D⟦fun I(I1;E)⟧r u s = fbinding(I,I1,E⟦E⟧)r u s
where
   fbinding:Ide×Ide×Cd→Env→Ed→Store→Ans
   fbinding(I,I1,d) r u s =
        u((λk e s.E⟦E⟧r[e/I1]k s)/I)s
D⟦D1 ; D2⟧ r u s = elaboration(D⟦D1⟧,D⟦D2⟧) r u s
where
   elaboration:Dd×Dd→Env→Dc→Store→Ans
   elaboration(d1,d2)r u s =
      d1 r(λr1 s.d2 r[r1] (λr2 s.u(r1[r2])s)s)s
```

Figure 7: Declaration Componentization

```
E⟦true⟧ r k s = value(true) r k s
E⟦false⟧ r k s = value(false) r k s
where
   value:Bool →Env→Ec→Store→Ans
   value b r k s = k b s
E⟦B⟧ r k s = value(number(B)) r k s
   where
   value:Num→Env→Ec→Store→Ans
   value n r k s = k n s
E⟦I⟧r k s = association(I) r k s
where
   association:Id→Env→Ec rarrow Store→Ans
   association(I) r k s =
      (r I = unbound) → error, k (r I) s
E⟦read⟧ r k s = read r k s
where
   read:  Env→Ec→Store→Ans
   read r k s = null(s input) → error,
           k (hd(s input)) s[tl(s input)/input]
E⟦E1 O E2⟧ r k s = operation(O,R⟦E1⟧,R⟦E2⟧)r k s
where
   operation:Opr × Ed × Ed→Env→Ec→Store→Ans
   operation(o,d1, d2) r k s =
       d1 r (λv1s.d2 r (λv2s.apply(o,v1,v2)k s) s) s
E⟦E1(E2)⟧r k s = fcall(E⟦E1⟧,E⟦E2⟧)r k s
where
  fcall:  Ed × Ed →Env→Cc→Store→Ans
  fcall(d1,d2)r k s = d1 r(Fun?(λf s.d2 r(f k)))s
E⟦if E then E1 else E2⟧r k s =
             choice(E⟦E⟧,E⟦E1⟧,E⟦E2⟧)r k s
where
  choice:  Ed×Ed×Ed →Env→Cc→Store→Ans
  choice(d,d1,d2) r k s = d r (λv s.Bool?(λv s.
         cond(d1 r k s,d2 r k s)v)v s)s
R⟦E⟧ r k s = dereference(E⟦E⟧) r k s
where
   dereference:Ed→Env→Store→Ans
   dereference(d) r k s =
       d r (λe s.deref(λv s.Rv?k e s)e s) s
```

Figure 8: Expression Componentization

5. RELATED WORK

The component-based style for denotational semantics is a complementary approach to other proposed solution to the legibility problem. For instance, the incremental definition style of Tirelo at alli[11], which is based on the linguist concept of vagueness can benefit from the use of components. In the incremental approach details are added stepwisely to a simpler definition by means of a mechanism named denotation transformation. The use of components may help separating concerns, which is very important to facilitate the integration of new elements to the definition.

Another important attempt to solve the legibility problem is the monadic semantics proposed by Moggi[4, 5]. This proposal also removes the context information from the equations and, consequently, reaches high level of modularity. However, monad semantics requires complex and intricate monad transformation operations. Component-based semantics are much simpler, they can encapsulate fundamental concepts in a way easy to use.

Apparently P. Mosses [8] has avoided the use of denotational semantics as the basis for a technique based on components due to the low legibility caused by the explicit use of context information. The present proposal overcomes these difficulties.

A comparison with other approaches to formal semantics, such as action semantics and structured operational semantics, is not addressed at this moment because the focus of this work is the improvement of the legibility of denotational semantics, not to make it supersede other models. Each formal method has its proper niche, in which it produces better results. Component-based denotational semantics just brings value to this formalism.

6. CONCLUSIONS

This work proposes an style to organizing denotational semantics description, inspired in P. Mosses's work [8, 9, 10] on action and structured operational semantics, in order to produce noticeable improvement in formal definition legibility.

A judicious use of denotational components permits the removal of context dependence from the presentation of semantic equations, making them more legible. To this purpose, the component signatures are standardized and the flow of context information is encapsulated within these components.

The result is that semantic definitions are reduced to a mapping from abstract syntax constructs to their denotations expressed as a combination of denotational components.

The encapsulation of fundamental and intricate concepts of programming languages may contribute to make formal semantics popular and turn the descriptions of the semantics of sizable programming languages readable by programmers and computer scientists.

The next step is to standardize the notion of context and to define a library of generic components that are applica-

```
C⟦E1 := E2⟧ r c s = assignment(E⟦E1⟧,R⟦E1⟧)r c s
where
  assignment:Ed × Ed →Env →Cc →Store →Ans
  assignment (d1, d2) r c s =
     d1 r( Loc?λl s.d2 r (λv s.update l c v s)s))s
C⟦output E⟧ r c s = write(R⟦E⟧) r c s
where
  write:Ed →Env →Cc →Store →Ans
  write(d) r c s = d r(λe s.(e,c s))s
C⟦if E then C1 else C2⟧r c s =
          choice(R⟦E⟧,C⟦C1⟧,C⟦C2⟧) r c s
where
   choice:Ed × Cd × Cd →Env →Cc →Store →Ans
   choice(b,d1,d2) r c s = b r (λv s.Bool?
            (λv s.cond(d1 r c s, d2 r c s)v)v s)s
C⟦while E do C⟧ r c s = loop(R⟦E⟧,C⟦C⟧)r c s
where
   loop:Ed×Cd →Env→Cc→Store→Ans
   loop(b,d) r c s = fix λf.  b r (Bool?
         (λv s.cond(d r(λs.f r c s)s,c s)v))s
C⟦E1(E2)⟧r c s = pcall(E⟦E1⟧,E⟦E2⟧)r c s
where
   pcall:  Ed × Ed →Env→Cc→Store→Ans
   pcall(d1,d2)r c s = d1 r(Proc?(λp s.d2r(p c)))s
C⟦C1 ; C2⟧ r c s = execution(C⟦C1⟧,C⟦C2⟧)r c s
where
   execution:Cd × Cd →Env →Store →Ans
   execution(d1,d2)r c s = d1 r(d2 r c)s
C⟦begin D ; C end⟧ r c s = block (D⟦D⟧, C⟦C⟧)r c s
where
  block:Dd × Cd →Env →Cc →Store →Ans
  block(e,d) r c s = e r (λr1 s.d r[r1] c s) s
```

Figure 9: Command Componentization

```
..... ..... ..... Programs ..... ..... .....
P⟦program C⟧ i = C⟦C⟧i r0 c0 s0
   where r0 = λi.unbound
         c0 = λs.(stop,s)
         s0 = (λl.unused)[i/input]
   N.B.:  input ∈ Loc is a reserved location
..... ..... ..... Declarations ..... ..... .....
D⟦const I = E⟧ = cbinding(I,R⟦E⟧)
D⟦var I = E⟧ = vbinding(I,R⟦E⟧)
D⟦proc I(I1;C)⟧ = pbinding(I,I1,C⟦C⟧)
D⟦fun I(I1;E)⟧ = fbinding(I,I1,E⟦E⟧
D⟦D1 ; D2⟧ = elaboration(D⟦D1⟧,D⟦D2⟧)
..... ..... ..... Expressions ..... ..... .....
E⟦true⟧ = value(true)
E⟦false⟧ = value(false)
E⟦B⟧ = value(number(B))
E⟦I⟧ = association(I)
E⟦read⟧ = read
E⟦E1 O E2⟧ = operation(O,R⟦E1⟧,R⟦E2⟧)
E⟦E1(E2)⟧ = fcall(E⟦E1⟧,E⟦E2⟧)
E⟦if E then E1 else E2⟧ = choice(E⟦E⟧,E⟦E1⟧,E⟦E2⟧)
R⟦E⟧ = dereference(E⟦E⟧)
..... ..... ..... Commands ..... ..... .....
C⟦E1 := E2⟧ = assignment(E⟦E1⟧, R⟦E2⟧)
C⟦output E⟧ = write(R⟦E⟧)
C⟦if E then C1 else C2⟧ = choice(R⟦E⟧,C⟦C1⟧,C⟦C2⟧)
C⟦while E do C⟧ = loop(R⟦E⟧,C⟦C⟧)
C⟦C1 ; C2⟧ = execution(C⟦C1⟧,C⟦C2⟧)
C⟦E1(E2)⟧ = pcall(E⟦E1⟧,E⟦E2⟧)
C⟦begin D ; C end⟧ = block(D⟦D⟧,C⟦C⟧)
```

Figure 10: Component Semantics of Small

ble to any programmig languages so that the construction of new descriptions could be simply an act of putting together predefined components from this library, without any concerns to context handling. A collateral effect of the use of generic denotational components is that formal descriptions may become scalable.

7. REFERENCES

[1] H. B. Curry and R. Feys. *Combinatory Logic I.* North-Holland, Amsterdam, 1958.

[2] Michael J. C. Gordon. *The denotational description of programming languages - an introduction.* Springer-Verlag, 1979.

[3] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[4] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

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

[6] Peter D. Mosses. The modularity of action semantics. Internal Report IR–75, Dept. of Computer Science, Univ. of Aarhus, 1988. Revised version of a paper presented at a CSLI Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California, March 1987 (proceedings unpublished).

[7] Peter D. Mosses. The varieties of programming language semantics. In *Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics: Akademgorodok, Novosibirsk, Russia*, volume 2244 of *PSI '02*, pages 165–190, London, UK, UK, 2001. Springer-Verlag.

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

[9] Peter D. Mosses. Component-based description of programming languages. In *BCS International Aca demic Conference 2008 ? Visions of Computer Science*, pages 275–286, 2008.

[10] Peter D. Mosses. Component-based semantics. In *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, SAVCBS '09, pages 3–10, New York, NY, USA, 2009. ACM.

[11] Fabio Tirelo, Roberto S. Bigonha, and João Saraiva. Disentangling denotational semantics definitions. *Journal of Universal Computer Science*, 14(21):3592–3607, dec 2008.

[12] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39:14–30, March 2004.