The Denotational Semantics of an Object Oriented Programming Language

Andreas V. Hense

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 01/90

The Denotational Semantics of an Object Oriented Programming Language

Andreas V. Hense*

February 1, 1990

Abstract

Recently, several descriptions of object oriented programming languages with denotational semantics have been given. Cook presented a denotational semantics of class inheritance. This semantics abstracts from the internal state of objects, which is one of their salient characteristics.

In this paper we show that Cook's denotational semantics of class inheritance is applicable to object oriented programming languages, where objects have a state. For this purpose we define a direct denotational semantics of a small object oriented programming language. We claim that the resulting denotational semantics is clear and can serve the derivation of efficient implementations of object oriented programming languages.

*Lehrstuhl für Programmiersprachen und Übersetzerbau, FB–Informatik, Universität des Saarlandes, 6600 Saarbrücken 11, Fed. Rep. of Germany, e-mail: hense@cs.uni-sb.de

1 Introduction

The cause for one major difficulty of understanding the important principles of object oriented programming lies in the lack of a clear denotational semantics. The novice to object oriented concepts is often the victim of the fallacious simplicity of the existing operational semantics and it is through a certain practical experience with a particular object oriented programming language that he suddenly grasps the true nature of object oriented programming languages and especially of class inheritance.

The first semantics for Smalltalk[1] was operational. [Wol87] described the semantics of a subset of Smalltalk in the denotational style. This semantics still has some operational elements: inheritance is described by method lookup. [Kam88] described Smalltalk with a denotational semantics in continuation style. Both semantics have the disadvantage of being long because they describe a large subset of Smalltalk. This disadvantage was removed by [Red88] who described a small object oriented programming language with a direct semantics. Like [Car84,Car88] he uses fixed points for modelling *self*. [CP89] described the semantics of inheritance without state. This results in a clear semantics. Cook tried to apply this semantics to an object oriented programming language [Coo89] but he did not come to convincing results. Thus it still remains to be shown that this semantics is applicable to an object oriented programming language.

The essential features of object oriented programming are listed in [Weg87]: an object oriented programming language must support the concepts of *objects*, *object classes*, and *class inheritance*. An object has a set of operations and a state. Objects communicate with each other by message sending. The result of a message sent to an object (the receiver) is not completely determined by the actual parameters but depends on the state of the receiver. Object classes specify an interface of operations. They can serve as templates for creating objects with the specified interface. They may also contain the implementation of the operations specified in the interface. Class inheritance is a mechanism for the composition of interfaces. There is dynamic binding[2] for operations modified in subclasses.

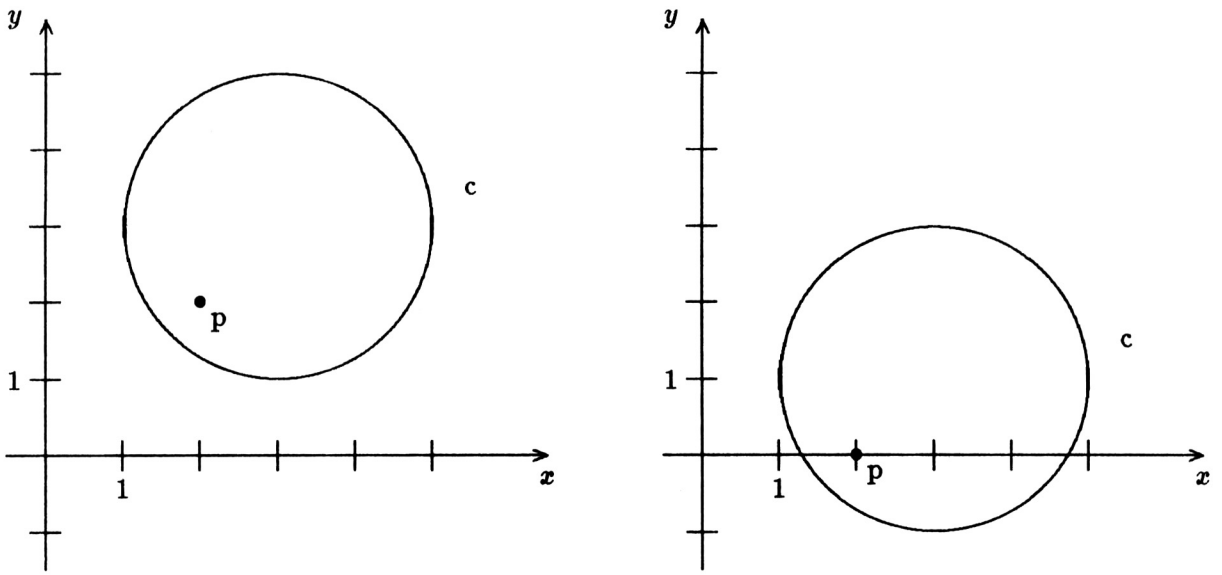
The programming language O'small, which is presented in this paper, is an object oriented programming language in this sense. The name O'small gives a hint at the purpose and the origin of the language. [Gor79] described an imperative programming language called SMALL by giving it a denotational semantics. O'small is an object oriented extension of SMALL. The reason for describing an extension of a well known programming language concept instead of describing the prototype Smalltalk is "inheritance at another level": we only show the differences between object oriented programming languages and imperative programming languages; the description of the latter is well known [Sto77,Gor79].

1.1 Overview

In anticipation of section 4 the next section intuitively introduces our object oriented programming language O'small. Section 3 describes the semantics of inheritance in method systems. Method systems are an abstraction of object oriented programming languages. In method systems there is no state. In section 4 we present the abstract syntax and the direct semantics of O'small. This semantics is based on the semantics of inheritance in method systems. In other words: the semantics of inheritance in method systems is embedded in a denotational semantics without continuations (direct semantics). In section 4.6 O'small is extended to allow the creation of an object of the receiver's class.

[1] Smalltalk [GR89] serves as a prototype of object oriented programming languages.

[2] this concept is also called late binding



before moving after moving

Figure 1: Points and circles in the plain

2 An Introductory Example

Before we present the syntax and the semantics of our object oriented programming language we will give an intuitive introduction by an example. The O'small program in figure 2 on page 4 is derived from an example by [CP89][3]: it is about points and circles with Cartesian coordinates in the plain. Points and circles can be moved in the plain as in figure 1.

There are two class definitions and the inheritance graph is as in figure 3. The class *Base* is a class "without contents" (see section 4.5). Objects of class *Point* have two instance variables representing the Cartesian coordinates of the point. A point object created with *new* is in the origin because its instance variables are initialized with zero. There are two functions for inspecting the instance variables; instance variables are not visible from outside the instance. The procedure *move* changes the position of the receiver. In object oriented terminology the O'small expression[4] $p.m(a)$ stands for the sending of m with argument a to the receiver p. There is a function for the distance from the origin and a function that returns TRUE if the receiver is closer to the origin than the argument. Booleans and numbers are primitive with some standard functions on them to keep the example concise.

We use method as the generic term for procedures and functions of O'small. The class *Circle*, which inherits instance variables and methods from *Point*, has an additional instance variable for the radius, a methods for reading and changing the radius, and it redefines the distance-function. For the redefinition of the distance-function the distance-function of the superclass is referred to with *super.distFromOrg*. Note that the inherited function *closerToOrg* was not redefined in the class *Circle*. Nevertheless with *self.distFromOrg* in the body of *closerToOrg* the distance-function of *Circle* is meant when the receiver of *closerToOrg* is in class *Circle* (dynamic binding). The output of example 2 results in: FALSE FALSE. This is what we intended. We are now able to compare points and circles with respect to their closeness to the origin and always get consistent behavior.

[3]The concrete syntax of O'small has been developed for test purposes and does not claim to be aesthetically pleasing.

[4]or command

```
def class Point inherit Base
    private var xComp := 0; var yComp := 0
    in fun x() return xComp;
       fun y() return yComp;
       proc move(X,Y) xComp := X+xComp; yComp := Y+yComp end;
       fun distFromOrg() return  sqrt(xComp*xComp + yComp*yComp);
       fun closerToOrg(point) return
          self.distFromOrg < point.distFromOrg
    end;
    class Circle inherit Point
    private var radius := 0
    in fun r() return radius;
       proc setR(r) radius := r end;
       fun distFromOrg() return max(0, super.distFromOrg - radius)
    end
in
    def var p := new Point;
        var c := new Circle
    in p.move(2,2); c.move(3,3); c.setR(2);
       output p.closerToOrg(c);

       p.move(0,-2);     c.move(0,-2);          | moving |

       output p.closerToOrg(c)
    end
end
```

Figure 2: Example program in O'small

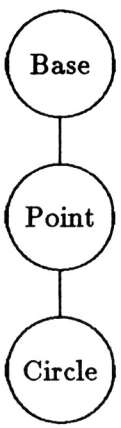


Figure 3: Inheritance graph

3 Semantics of Inheritance

The contents of this section are taken from [Coo89]. Note that this semantics abstracts from state. So it is *not* a description of O'small, which was introduced informally by example 2. The semantics of O'small will be introduced in section 4.

Definition 1 A *record* is a finite mapping from a set of labels onto a set of values. A record is denoted by: $[x_1 \rightarrow v_1, \cdots, x_n \rightarrow v_n]$ where the x_i are labels and the v_i are values. All labels that are not in the list are mapped onto $\perp$.

An *object* is a record with functions as values. A *generator* is a function to which a fixed point operator can be applied resulting in an object. Its first formal parameter represents self-reference. The functional for the factorial function is an example of a generator:

$$FACT \;=\; \lambda s.\lambda n.\; \mathit{if}\; n = 1\; \mathit{then}\; 1\; \mathit{else}\; n * s(n-1)$$

Let $\mathbf{Y}$ denote the fixed point operator. For the fixed point operator the following holds: $\mathbf{Y}f = f(\mathbf{Y}f)$. The factorial function *fact* is defined as the least fixed point of the generator $FACT$:

$$\mathit{fact} \;=\; \mathbf{Y}(FACT)$$

A *class* is a generator which creates objects. C is an example of a class:

$$C \;=\; \lambda s.\,[\mathit{sqr} \;\rightarrow\; \lambda x.x * x,\;\; \mathit{abs} \;\rightarrow\; \lambda x.\sqrt{s.\mathit{sqr}(x)}]$$

If, as in this case, a class has no further parameters the objects it creates are all identical. The object c is created by application of the fixed point operator to the class C:

$$\begin{aligned} c \;&=\; \mathbf{Y}(C) \\ &=\; [\mathit{sqr} \;\rightarrow\; \lambda x.x * x,\;\; \mathit{abs} \;\rightarrow\; \lambda x.\sqrt{\mathbf{Y}(C).\mathit{sqr}(x)}] \\ &=\; [\mathit{sqr} \;\rightarrow\; \lambda x.x * x,\;\; \mathit{abs} \;\rightarrow\; \lambda x.\sqrt{x * x}] \end{aligned}$$

Inheritance is the derivation of a new generator from an existing one[5], whereby the formal parameters for self-reference of both generators are shared. A *wrapper* is a function that modifies a generator in a self-referential way. A wrapper has a parameter for self-reference and a parameter for the generator it modifies.

Definition 2 Let $dom(m) = \{x \mid m(x) \neq \perp\}$. The *left-preferential combination of records* is defined by:

$$(m \oplus n)(s) = \begin{cases} m(s) & \text{if } s \in dom(m) \\ n(s) & \text{if } s \in dom(n) - dom(m) \\ \perp & \text{otherwise} \end{cases}$$

A *record wrapper* is a binary function on records with the first argument for self-reference and the second argument for the record to modify.

Definition 3 Let $*$ be a binary operator on values. The *distributive version* of $*$ is denoted by $\boxed{*}$. It operates on generators and is defined by:

$$G_1 \boxed{*} G_2 = \lambda s.G_1(s) * G_2(s)$$

[5] for multiple inheritance: from several existing generators.

Definition 4 The *inheritance function* $\boxed{\triangleright}$ applies a record wrapper to a class generator and returns a class generator. It is defined by:

$$W \boxed{\triangleright} G = (W \boxed{\cdot} G) \boxed{\oplus} G$$

where $w \triangleright g = (w \cdot g) \oplus g = w(g) \oplus g$ and $\cdot$ is the application.

Wrappers are central to the semantics of inheritance. In every class declaration a superclass is named. If nothing is inherited *Base* is named as superclass. The semantics of class *Base* is a class generator. The semantics of a class definition is a wrapper being wrapped around the class generator of the superclass, and this results in a new class generator. As pointed out above we can get objects by applying the fixed point operator to a class generator. Examples of wrapper applications can be found in [Coo89].

Note that in this section objects have no state. The definitions presented so far will be used in section 4. In section 4 objects have a state. The definitions are also used in [CP89] for the definition of method systems.

3.1 Method Systems

Method systems are a model of object oriented programming languages. Method systems describe inheritance only. They do *not* describe state, i.e. there are no assignments, no instance variables etc. In [CP89] a *denotational semantics* for method systems based on wrapper application is given. This denotational semantics is proved to be equivalent to a *method lookup semantics*, the operational semantics of inheritance in Smalltalk.

4 Semantics of Object Oriented Programming Languages

In section 3 we briefly reviewed the semantics of inheritance by [Coo89,CP89]. [Coo89] then goes on and builds a continuation style semantics for a small object oriented programming language. We found this semantics difficult to verify. In addition, when continuation style semantics is not necessary, we prefer direct semantics because the latter is easier to understand.

In this section we show how to extend the semantics of inheritance without state of section 3 to a semantics of an object oriented programming language, i.e. we add the state we abstracted from in section 3. The proper semantics definition consists of the abstract syntax (section 4.4) and the mapping from the syntactic domains onto the semantic domains defined by the semantic clauses (section 4.5).

4.1 Designing O'small

We designed the programming language O'small for the semantics description of object oriented programming languages. It is based on SMALL [Gor79], an imperative programming language for semantics description. O'small is limited in general, among others it does not include potentially infinite data structures, but it contains all the essential features of an object oriented programming language (section 1). For the formulation of examples O'small is provided with a concrete syntax. The description of the concrete syntax is not included in this paper.

Some Properties of O'small are listed now. A class definition consists of a private clause, where the instance variables are declared, and a method clause. Instance variables are not visible outside the object. Method definitions are restricted to the method clause of class definitions. Therefore methods can only be called via message sending. Instance variables are encapsulated: they are only accessible to methods defined in the class but not to methods defined in a subclass. Class and variable definitions figure in the same syntactic clause and thus nested class definitions

become possible. This feature may be subject to discussion. There is "call by reference" for parameters of functions and procedures. After an assignment $x := y$, where x and y denote objects, x denotes the same object as y.

4.2 Extending the Semantic Domains

In the description of the imperative programming language SMALL there are three semantic domains for values. For the description of O'small these domains have to be extended. There are *Storable values* which can be put into locations in the store. *Denotable values* can be bound to an identifier in an environment. *Expressible values* can be the result of expressions. Storable values are so called R-values and files. Files serve for input and output. R-values are the results of evaluating the right hand sides of assignments. We extend R-values by objects. Denotable values are locations in the store, R-values, procedures and functions. We extend the denotable values by class generators and wrappers. Expressible values are the same as denotable values. We will refer to them as denotable values from now on.

4.3 The New Semantic Domains

The new semantic domains by which we extend storable and denotable values are *Obj*, *ClGen*, and *Wrapper*. Objects, class generators, and wrappers were introduced in section 3. Their domains were:

$$\begin{array}{lcl} \textbf{Obj} & = & \textbf{Rec} \\ \textbf{ClGen} & = & \textbf{Obj} \rightarrow \textbf{Obj} \\ \textbf{Wrapper} & = & \textbf{Obj} \rightarrow \textbf{Obj} \rightarrow \textbf{Obj} \end{array}$$

What are the new semantic domains in the semantics of O'small which correspond to the semantics of inheritance? For *Obj* it is still *Rec* (although the record values are different now). The domain of wrappers is completely determined by the domain of class generators and the latter is discussed now.

The domain of class generators

To understand the domain of class generators we take a closer look at class declaration and object creation. When a class is declared, the current environment is enriched by the class name. The class name is bound to the result of a wrapper application. In this wrapper application the wrapper for the current class is applied to the class generator of the superclass. The store remains unchanged because the instance variables are not allocated at the time of the class declaration. An object is created by application of the fixed point operator to the class generator. For the fixed point operator to be applied to it the domain of the class generator must be

$$\alpha \rightarrow \alpha$$

where α is any domain (the domain of class generators was $Obj \rightarrow Obj$ in section 3). The environment for methods is recursive whereas the environment for instance variables is not. Therefore we allocate the instance variables after the application of the fixed point operator. A function is needed for the allocation of all instance variables[6] of the new object. This function has to "know" the current store and has to return it with the instance variables inside it; the store must thus appear in the domain and the codomain of the function. In addition this function has to return an object. Therefore the result of the application of the fixed point operator to the class generator is:

$$\textbf{Store} \rightarrow (\textbf{Obj} \times \textbf{Store})$$

[6]including the instance variables declared in superclasses

This is our α. Thus the *domain for class generators* is:

$$(\mathbf{Store} \to (\mathbf{Obj} \times \mathbf{Store})) \to (\mathbf{Store} \to (\mathbf{Obj} \times \mathbf{Store}))$$

4.4 Syntax of O'small

Our way of describing semantics goes back to [Sto77] and [Gor79].

4.4.1 Syntactic Domains

Primitive syntactic domains

Ide	the domain of identifiers	**I**
Bas	the domain of basic constants	**B**
BinOp	the domain of binary operators	**O**

To the right are the variables referring to elements of the respective domain in the clauses.

Compound syntactic domains

Pro	the domain of programs	**P**
Exp	the domain of expressions	**E**
Com	the domain of commands	**C**
VaCl	the domain of variable and class declarations	**V**
Meth	the domain of method declarations	**F**

Method declarations are distinguished from variable and class declarations because methods are only declared in classes.

4.4.2 Syntactic Clauses

P ::= **program C**

E ::= **B** | **true** | **false** | **read** | **I** | **E.I(E$_1$,...,E$_n$)** |
new E | **def V in E** | **if E then E$_1$ else E$_2$** | **E$_1$ O E$_2$**

C ::= **E.I(E$_1$,...,E$_n$)** | **I := E** | **output E** | **if E then C$_1$ else C$_2$** |
while E do C | **def V in C** | **C$_1$;C$_2$**

V ::= **var I := E** | **class I$_1$ inherit I$_2$ private V in M** | **V$_1$; V$_2$**

M ::=**proc I(I$_1$,...,I$_n$) C** | **fun I(I$_1$,...,I$_n$) local V in C return E** | **F$_1$; F$_2$**

Functions may declare local variables and contain a series of commands. The expression after **return** is compulsory.

4.5 Semantics of O'small

4.5.1 Semantic Domains

Primitive semantic domains:

Bool	the domain of booleans	**b**
Loc	the domain of locations	**i**
Bv	the domain of basic values	**e**

Compound semantic domains are defined by the following domain equations:

Rec	=	**Ide** $\rightarrow$ **[Dv + {unbound}]**	records	**r**
Env	=	**Rec**	environments	**r**
Obj	=	**Rec**	objects	**o**
Dv	=	**Loc + Rv +** $\mathbf{Proc}_n$ **+** $\mathbf{Fun}_n$		
		ClGen + Wrapper	denotable values	**d**
Sv	=	**File + Rv**	storable values	**v**
Rv	=	**Bool + Bv + Obj**	R-values	**e**
File	=	$\mathbf{Rv}^*$	files	**i**
Store	=	**Loc** $\rightarrow$ **[Sv + {unused}]**	stores	**s**
$\mathbf{Proc}_n$	=	$\mathbf{Dv}^n$ $\rightarrow$ **Store** $\rightarrow$ **Store**	procedure values	**p**
$\mathbf{Fun}_n$	=	$\mathbf{Dv}^n$ $\rightarrow$ **Store** $\rightarrow$ **[Ev**$\times$**Store]**	function values	**f**
ClGen	=	**Fixed** $\rightarrow$ **Fixed**	class generators	**g**
Fixed	=	**Store** $\rightarrow$ **[Obj**$\times$**Store]**	result of a fixed point construction	**x**
Wrapper	=	**Fixed** $\rightarrow$ **ClGen**	wrappers	**w**
Ans	=	**File**$\times${**error, stop**}	program answers	**a**

Domains $\mathbf{Proc}_n$ and $\mathbf{Fun}_n$ are needed for each $n \in \mathbb{N}_0$.

4.5.2 Semantic Clauses

The following semantic functions are primitive:

B : **Bas** $\rightarrow$ **Bv**

O : **BinOp** $\rightarrow$ **Rv** $\rightarrow$ **Rv** $\rightarrow$ **Store** $\rightarrow$ **[Dv**$\times$**Store]**

B takes syntactic basic constants and returns semantic basic values. O takes a syntactic binary operator (e.g. +), two R-values, and a store; it returns the result of the binary operation and, in general a changed store. The remaining semantic functions will be defined by clauses and have the following types:

$$
\begin{array}{ll}
\mathsf{P} & : \mathbf{Pro} \to \mathbf{File} \to \mathbf{Ans} \\
\mathsf{R},\ \mathsf{E} & : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Store} \to [\mathbf{Dv} \times \mathbf{Store}] \\
\mathsf{C} & : \mathbf{Com} \to \mathbf{Env} \to \mathbf{Store} \to \mathbf{Store} \\
\mathsf{V} & : \mathbf{VaCl} \to \mathbf{Env} \to \mathbf{Store} \to [\mathbf{Env} \times \mathbf{Store}] \\
\mathsf{M} & : \mathbf{Meth} \to \mathbf{Env} \to \mathbf{Env}
\end{array}
$$

Definitions

$$
\begin{array}{l}
\mathsf{P}[\![\textbf{program C}]\!]\textbf{i} = \textbf{extractans } s_{final} \\
\quad \textbf{where} \\
\quad \textbf{extractans} = \lambda \textbf{s.(s out, (s err} \to \textbf{error, stop))} \\
\quad s_{final} = \mathsf{C}[\![\mathbf{C}]\!]\mathbf{r}_{initial}\ s_{initial} \\
\quad \mathbf{r}_{initial} = (\lambda \textbf{I.unbound})\ [\ (\lambda \textbf{os.}(\lambda \textbf{I.unbound,s}))\ /\ \textbf{Base}\ ] \\
\quad s_{initial} = (\lambda \textbf{l.unused})\ \textbf{[false/err, i/inp, } \epsilon \textbf{/out]}
\end{array}
$$

An answer from a program is gained by running it with an input. The store is initialized with the error flag set to **false**, the input, and an empty output. The initial environment contains a class generator **Base**. Objects of the base class are records where every label is mapped to $\perp$. In addition to the output the error flag shows if the program has come to a normal end (**stop**) or if it stopped with an error (**error**). For the definition of auxiliary functions in the following clauses refer to appendix A.

$$
\mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} = \mathsf{E}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{deref} \star \textbf{Rv?}
$$

The semantic function R produces R-values.

$$
\begin{array}{lcl}
\mathsf{E}\ [\![\mathbf{B}]\!]\mathbf{r} & = & \textbf{result(B}[\![\mathbf{B}]\!]\textbf{)} \\
\mathsf{E}\ [\![\textbf{true}]\!]\mathbf{r} & = & \textbf{result true} \\
\mathsf{E}\ [\![\textbf{false}]\!]\mathbf{r} & = & \textbf{result false} \\
\mathsf{E}\ [\![\textbf{read}]\!]\mathbf{r} & = & \textbf{cont inp} \star \lambda \textbf{is. (}\ \textbf{i=}\epsilon \to \textbf{seterr s, (hd i, s[tl i/inp])}\ \textbf{)} \\
\mathsf{E}\ [\![\mathbf{I}]\!]\mathbf{r} & = & \textbf{result (r I)} \star \textbf{Dv?} \\
\mathsf{E}\ [\![\mathbf{E.I(E}_1\textbf{,...,E}_n\textbf{)}]\!]\mathbf{r} & = & \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{Obj?} \star \\
 & & \lambda \textbf{o.(result(o I)}\star\textbf{Fun?}\star\lambda \textbf{f.}\mathsf{R}[\![\mathbf{E}_1]\!]\mathbf{r}\star\lambda \mathbf{d}_1\textbf{.}\ \ldots \mathsf{R}[\![\mathbf{E}_n]\!]\mathbf{r}\star\lambda \mathbf{d}_n\textbf{.f(}\mathbf{d}_1\textbf{,}\ldots\textbf{,}\mathbf{d}_n\textbf{))}
\end{array}
$$

The last clause is for message sending, which is record field selection (hence the notation). The first expression is evaluated as an R-value. The result of this evaluation must be an object. The resulting record **o** is applied to the message **I**. This should result in a function that is then

applied to the parameter.

$$\mathsf{E}[\![\textbf{new E}]\!]\mathbf{r} \;=\; \mathsf{E}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{ClGen?} \star \lambda\mathbf{gs.(Y\ g)s}$$

After evaluating **E** we get a class generator. The fixed point operator **Y** is applied to this class generator. The result of the application of **Y** is applied to the current store **s**.

$$\begin{aligned}
\mathsf{E}[\![\textbf{def V in E}]\!]\mathbf{r} &= \mathsf{V}[\![\mathbf{V}]\!]\mathbf{r} \star \lambda\mathbf{r'}.\mathsf{E}[\![\mathbf{E}]\!]\mathbf{r}[\mathbf{r'}] \\
\mathsf{E}[\![\textbf{if E then } \mathbf{E_1} \textbf{ else } \mathbf{E_2}]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{Bool?} \star \textbf{cond}(\mathsf{E}[\![\mathbf{E_1}]\!]\mathbf{r}, \mathsf{E}[\![\mathbf{E_2}]\!]\mathbf{r}) \\
\mathsf{E}[\![\mathbf{E_1}\ \mathbf{O}\ \mathbf{E_2}]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E_1}]\!]\mathbf{r} \star \lambda\mathbf{e_1}.\mathsf{R}[\![\mathbf{E_2}]\!]\mathbf{r} \star \lambda\mathbf{e_2}.\mathsf{O}[\![\mathbf{O}]\!](\mathbf{e_1}, \mathbf{e_2})
\end{aligned}$$

$$\begin{aligned}
\mathsf{C}[\![\mathbf{E.I}(\mathbf{E_1},\ldots,\mathbf{E_n})]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{Obj?} \star \lambda\mathbf{o.result(o\ I)}\star\textbf{Proc?} \\
&\quad \star \lambda\mathbf{p}.\mathsf{R}[\![\mathbf{E_1}]\!]\mathbf{r} \star \lambda\mathbf{d_1}.\ \ldots\mathsf{R}[\![\mathbf{E_n}]\!]\mathbf{r}\star\lambda\mathbf{d_n}.\mathbf{p}(\mathbf{d_1},\ldots,\mathbf{d_n}) \\
\mathsf{C}[\![\mathbf{I} := \mathbf{E}]\!]\mathbf{r} &= \mathsf{E}[\![\mathbf{I}]\!]\mathbf{r} \star \textbf{Loc?} \star \lambda\mathbf{l}.\mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{(update l)} \\
\mathsf{C}[\![\textbf{output E}]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \lambda\mathbf{es.s[append(s\ out,e)/out]} \\
\mathsf{C}[\![\textbf{if E then } \mathbf{C_1} \textbf{ else } \mathbf{C_2}]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{Bool?} \star \textbf{cond}(\mathsf{C}[\![\mathbf{C_1}]\!]\mathbf{r}, \mathsf{C}[\![\mathbf{C_2}]\!]\mathbf{r}) \\
\mathsf{C}[\![\textbf{while E do C}]\!]\mathbf{r} &= \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \textbf{Bool?} \star \\
&\quad \textbf{cond}(\mathsf{C}[\![\mathbf{C}]\!]\mathbf{r} \star \mathsf{C}[\![\textbf{while E do C}]\!]\mathbf{r}, \lambda\mathbf{s.s}) \\
\mathsf{C}[\![\textbf{def V in C end}]\!]\mathbf{r} &= \mathsf{V}[\![\mathbf{V}]\!]\mathbf{r} \star \lambda\mathbf{r'}.\mathsf{C}[\![\mathbf{C}]\!]\mathbf{r}[\mathbf{r'}] \\
\mathsf{C}[\![\mathbf{C_1}\ ;\ \mathbf{C_2}]\!]\mathbf{r} &= \mathsf{C}[\![\mathbf{C_1}]\!]\mathbf{r} \star \mathsf{C}[\![\mathbf{C_2}]\!]\mathbf{r}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{V}[\![\textbf{class } \mathbf{I}_{self} \textbf{ inherit } \mathbf{I}_{super} \textbf{ private V in M}]\!]\mathbf{r} \\
&= \mathsf{E}[\![\mathbf{I}_{super}]\!]\mathbf{r} \star \textbf{ClGen?} \star \lambda\mathbf{g}_{super}.\textbf{result}(\mathbf{r}[\mathbf{w} \boxed{\triangleright} \mathbf{g}_{super}/\mathbf{I}_{self}]) \\
&\textbf{where} \\
&\mathbf{w} = \lambda\mathbf{x}_{self}\ \mathbf{x}_{super}\ \mathbf{s}_{create}.(\mathsf{M}[\![\mathbf{M}]\!]\mathbf{r}\ \mathbf{r}_{local}\ [\mathbf{r}_{self}/\textbf{self}, \mathbf{r}_{super}/\textbf{super}], \mathbf{s}_{new}) \\
&\qquad \textbf{where} \\
&\qquad (\mathbf{r}_{super}, \mathbf{s}_{super}) = \mathbf{x}_{super}\ \mathbf{s}_{create} \\
&\qquad (\mathbf{r}_{local}, \mathbf{s}_{new}) = \mathsf{V}[\![\mathbf{V}]\!]\mathbf{r}\ \mathbf{s}_{super} \\
&\qquad (\mathbf{r}_{self}, _) = \mathbf{x}_{self}\ \mathbf{s}_{new}
\end{aligned}$$

The result of the evaluation of the last clause is the binding of a class generator to the class name. The store remains unchanged when a class is declared. The wrapper **w** takes a class generator for self reference, a class generator for reference to the superclass, and a store as parameters. The store parameter is fed at object creation time, $\mathbf{x}_{self}$ is fed at the fixed point operation, and $\mathbf{x}_{super}$ is fed at the wrapper application. The wrapper evaluates the function and procedure definitions in an environment being determined at declaration time – except that the locations for the instance variables have to be determined at object creation time. The local environment is only visible in the class itself. Thus we have encapsulated instance variables.

In the inner where-clause above, environments and stores are created "successively". The instance variables for the superclass are allocated first. The instance variables for the current

class are allocated by evaluating the declarations **V** in the changed store. Then $\mathbf{x}_{self}$ is again applied to the changed store to give the self-environment. The careful reader may have noticed that the resulting store is not needed. This is indicated by an underlining. The reason for this is that the instance variables of the current class have been allocated already. The method environment is recursive, the instance variable environment is not.

$$\mathsf{V}[\![\textbf{var I := E}]\!]\mathbf{r} = \mathsf{R}[\![\mathbf{E}]\!]\mathbf{r} \star \lambda\mathbf{d.new} \star \lambda\mathbf{ls.(l/I,\ s[d/l])}$$

$$\mathsf{V}[\![\mathbf{V_1\ ;\ V_2}]\!]\mathbf{r} = \mathsf{V}[\![\mathbf{V_1}]\!]\mathbf{r} \star \lambda\mathbf{r'.V}[\![\mathbf{V_2}]\!]\mathbf{r[r']}$$

$$\mathsf{M}[\![\textbf{proc I}(\mathbf{I_1},\ldots,\mathbf{I_n})\ \mathbf{C}]\!]\mathbf{r} = \mathbf{r[p/I]}$$

$$\textbf{where p} = \lambda\mathbf{d_1}\ldots\mathbf{d_n.C}[\![\mathbf{C}]\!]\mathbf{r}[\mathbf{d_1/I_1},\ldots,\mathbf{d_n/I_n}]$$

$$\mathsf{M}[\![\textbf{fun I}(\mathbf{I_1},\ldots,\mathbf{I_n})\ \textbf{local V in C return E}]\!]\mathbf{r} = \mathbf{r[f/I]}$$

$$\textbf{where f} = \lambda\mathbf{d_1}\ldots\mathbf{d_n.V}[\![\mathbf{V}]\!]\mathbf{r}\star\lambda\mathbf{r'.C}[\![\mathbf{C}]\!]\mathbf{r[r']}[\mathbf{d_1/I_1},\ldots,\mathbf{d_n/I_n}]\star\mathsf{E}[\![\mathbf{E}]\!]\mathbf{r[r']}[\mathbf{d_1/I_1},\ldots,\mathbf{d_n/I_n}]$$

$$\mathsf{M}[\![\mathbf{M_1\ ;\ M_2}]\!]\mathbf{r} = (\mathsf{M}[\![\mathbf{F_1}]\!]\mathbf{r})\ [\mathsf{M}[\![\mathbf{F_2}]\!]\mathbf{r}]$$

Method definitions are not recursive. Recursion and the calling of other methods is made possible by sending messages to **self**.

4.6 Creating Objects of a Class Inside this Class

In Smalltalk it is possible to create a new instance of a class A inside class A, i.e. inside methods defined in class A, by the expression *self class new*. Let *self class new* occur in the definition of the method m defined in class A. Let B be a subclass of A where m is inherited without being redefined. Then the expression *self class new*, sent to an object b of class B, will return an object of class B. [Coo89] shows that an additional level of inheritance is needed to describe the possibilities of a Smalltalk-expression like *self class new*, where the class constructor (i.e. the class name) is referred to "in a relative way".

For the "relative" reference to the class constructor inside the class we extend O'small by the pseudo variable *current*. *current* denotes the class of the receiver of a message. Thus *current new* in O'small has the same effect as *self class new* in Smalltalk. Note that in O'small neither *current* nor *new* are messages; they have to be defined in the semantics. We need an Additional Level of Inheritance.

For this two clauses[7] of the semantics and the types of wrappers and class generators have to be modified.

$$\mathsf{V}[\![\textbf{class } \mathbf{I}_{self} \textbf{ inherit } \mathbf{I}_{super} \textbf{ private V in M}]\!]\mathbf{r}$$
$$= \mathsf{E}[\![\mathbf{I}_{super}]\!]\mathbf{r} \star \textbf{ClGen?} \star \lambda\mathbf{g}_{super}\textbf{.result}(\mathbf{r[w}\ \boxed{\triangleright}\ \mathbf{g}_{super}/\mathbf{I}_{self}])$$
$$\textbf{where}$$
$$\mathbf{w} = \lambda\mathbf{z}_{self}\ \mathbf{x}_{self}\ \mathbf{x}_{super}\ \mathbf{s}_{create}\textbf{.}(\mathsf{M}[\![\mathbf{M}]\!]\mathbf{r}\ \mathbf{r}_{local}\ [\mathbf{r}_{self}/\textbf{self},\ \mathbf{r}_{super}/\textbf{super},\ \lambda\mathbf{z.z}_{self}/\textbf{current}],\ \mathbf{s}_{new})$$
$$\textbf{where } \ldots$$

current contains the class generator being distributed to the wrapper and the generator of the superclass. The domains of class generators and wrappers have to be modified:

[7] and in the clause of the semantic function P replace λos by λzos.

$\mathbf{ClGen_{old}} =$	$\mathbf{Fixed \to Fixed}$	class generators as in 4.5.2	**z**
$\mathbf{ClGen} =$	$\mathbf{ClGen_{old} \to Fixed \to Fixed}$	class generators	**g**
$\mathbf{Wrapper} =$	$\mathbf{ClGen_{old} \to Fixed \to ClGen_{old}}$	wrappers	**w**

Now the fixed point operator has to be applied twice to a class generator when a new object is created.

$$\mathbf{E[\![new\ E]\!]r = E[\![E]\!]r \star ClGen? \star \lambda gs.(Y(Y\ g))s}$$

5 Conclusion

We have started with a semantics of inheritance as presented in [CP89]. This semantics does not include a state, and the question if it is applicable to an object oriented programming language with state remained open in [Coo89]. With our denotational semantics for O'small we showed that the semantics of inheritance is applicable to object oriented programming languages. For the description of objects with state the domains of class generators and wrappers had to be adjusted accordingly. It was shown that besides continuation style semantics also direct semantics is an appropriate way of describing object oriented programming languages. An additional level of inheritance to describe the creation of new objects of the receiver's class can be added easily. O'small is executable because we implemented its semantics in the functional programming language Miranda [Tur85].

5.1 Future Work

One direction of future research with our semantics is the discussion of different known concepts of object oriented programming languages (multiple inheritance, classes as objects, etc.). Another direction is the development of an efficient and provably correct implementation.

Acknowledgements
Thanks to Gerhard Hense and Christian Neusius for commenting on drafts of my paper. I am grateful to Andreas Gündel, Reinhold Heckmann, Fritz Müller, and Reinhard Wilhelm for ideas, constructive criticism, and helpful advice.

References

[Car84] Luca Cardelli. A semantics of multiple inheritance. *Lecture Notes in Computer Science*, 173:51–67, 1984.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance.* Technical Report CS-89-33, Brown University, Dept. of Computer Science, Providence, Rhode Island 02912, May 1989.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming Systems, Languages and Applications*, pages 433–444, ACM, October 1989.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, New York/Heidelberg/Berlin, 1979.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: the Language.* Addison-Wesley, 1989.

[Kam88] Samuel Kamin. Inheritance in Smalltalk-80. In *Symposium on Principles of Programming Languages*, pages 80–87, ACM, January 1988.

[Red88] U. S. Reddy. Objects as closures: abstract semantics of object-oriented languages. In *Symposium on Lisp and Functional Programming*, pages 289–297, ACM, 1988.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT press, 1977.

[Tur85] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. *Lecture Notes in Computer Science*, 201:1–16, 1985. Functional Programming Languages and Computer Architecture.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, pages 479–560, MIT Press, 1987.

[Wol87] Mario Wolczko. Semantics of smalltalk-80. In *ECOOP'87 European Conference on Object Oriented Programming*, pages 119–131, Paris, France, 1987.

A Auxiliary Functions

We need a generic function $\star$ for the composition of commands and declarations. This function stops the execution of the program when an error occurs. Let there be two functions f and g with the following types:

$$f : \left\{ \begin{array}{c} \mathbf{Store} \\ \mathbf{D_1} \to \mathbf{Store} \end{array} \right\}_a \to \left\{ \begin{array}{ll} \mathbf{Store}, & g : \mathbf{Store} \\ [\mathbf{D_2} \times \mathbf{Store}], & g : \mathbf{D_2} \to \mathbf{Store} \end{array} \right\}_b \to \left\{ \begin{array}{c} \mathbf{Store} \\ [\mathbf{D_3} \times \mathbf{Store}] \end{array} \right\}_c$$

The lines in braces represent alternatives. The alternatives in the following text are not free but depend on the choices of the three alternatives above: If above in the braces marked with an 'a' you choose the upper alternative, you have to choose the upper alternative in every brace marked with an 'a' below. If above in the braces marked with an 'a' you choose the lower alternative, you have to choose the lower alternative in every brace marked with an 'a' below. The same holds for the braces marked with 'b' or 'c'. Then the composition of f and g has type

$$f \star g : \left\{ \begin{array}{c} \mathbf{Store} \\ \mathbf{D_1} \to \mathbf{Store} \end{array} \right\}_a \to \left\{ \begin{array}{c} \mathbf{Store} \\ \mathbf{D_3} \to \mathbf{Store} \end{array} \right\}_c$$

and is defined by

$$f \star g = \left\{ \begin{array}{c} \lambda s_1 \\ \lambda d_1 s_1 \end{array} \right\}_a . \mathit{let} \left\{ \begin{array}{c} s_2 \\ (d_2, s_2) \end{array} \right\}_b = \left\{ \begin{array}{c} f s_1 \\ f d_1 s_1 \end{array} \right\}_a \mathit{in}\ \ s_2\, \mathbf{err} \to \left\{ \begin{array}{c} s_2 \\ (\bot, s_2) \end{array} \right\}_c, \left\{ \begin{array}{c} g\ s_2 \\ g\ d_2\ s_2 \end{array} \right\}_b$$

$\star$ is left associative. In lieu of the left-preferential combination of records the left-preferential combination of *Fixed* has to be defined:

$$x_1 \oplus x_2 \;=\; \lambda s.(r_1 \oplus_{lpr} r_2, s') \;\mathit{where}\; (r_1, s') = x_1 s,\ (r_2, \_) = x_2 s$$

$\oplus_{lpr}$ is defined in definition 2. This is the only change of the inheritance function (definition 4). Here are further auxiliary functions. Let **D** be any semantic domain:

cond : **[D** × **D]** → **Bool** → **D** ..... Alternative
cond(d_1, d_2) = λb.b → d_1, d_2

cnt : **Loc** → **Store** → [[**Sv** + {**unused**}] × **Store**] ..... Contents of a location
cnt = λls.(s l, s)

cont : **Dv** → **Store** → [**Sv** × **Store**] ..... Contents of a location with domain checking
cont = **Loc?** ⋆ cnt ⋆ **Sv?**

D? : **D'** → **Store** → [**D'** × **Store**], with **D** ⊆ **D'** ..... Domain checking
D? = λd. (isD d → result d, seterr)

deref : **Dv** → **Store** → [**Dv** × **Store**] ..... Dereferencing
deref = λe. (isLoc e → cont e, result e)

new : **Store** → [**Loc** × **Store**] ..... Getting a new location in the store
new s = (l,s) or = (⊥, s[true/err])

If new s = (l,s) then s l = **unused** is guaranteed.

result : **D** → **Store** → [**D** × **Store**] ..... Side effect free evaluation
result d = λs.(d, s)

seterr : **Store** → [**D** × **Store**] ..... Setting the error flag
seterr = λs.(⊥, s[true/err])

update : **Loc** → **Dv** → **Store** → **Store** ..... Updating of a location
update l = **Sv?** ⋆ λes.s[e/l]