Wrapper Semantics of an Object Oriented Programming Language with State

Andreas V. Hense

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 14/90

(this report overwrites Nr. A 01/90)

Wrapper Semantics of an Object Oriented Programming Language with State

Andreas V. Hense*

July 5, 1990

Abstract

Recently, several descriptions of object oriented programming languages with denotational semantics have been given. Cook presented an intuitive 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 example language. The insertion of state into class definitions can be done before or after the related fixed point operations. The choice of the alternative considerably influences the semantic domains and clauses. We claim that despite the introduction of state the resulting denotational semantics is clear and intuitive.

*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 method-lookup-semantics of class inheritance [GR89], which includes the semantics of the pseudo-variables *self* and *super*, may look obvious and simple to the novice. But we conjecture that this simplicity is a fallacious one and that the true nature of inheritance, namely the difference between inheriting from a class and using a class, may be concealed.

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] he uses fixed points for modeling *self*. [CP89] described the semantics of inheritance without state. This results in a clear semantics because inheritance is elucidated by a special mechanism called wrapper.

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 for operations modified in subclasses.

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

In this paper we describe a full object oriented language using wrappers. Our goal is a concise semantics that preserves as much as possible of the intuitiveness of the wrapper mechanism.

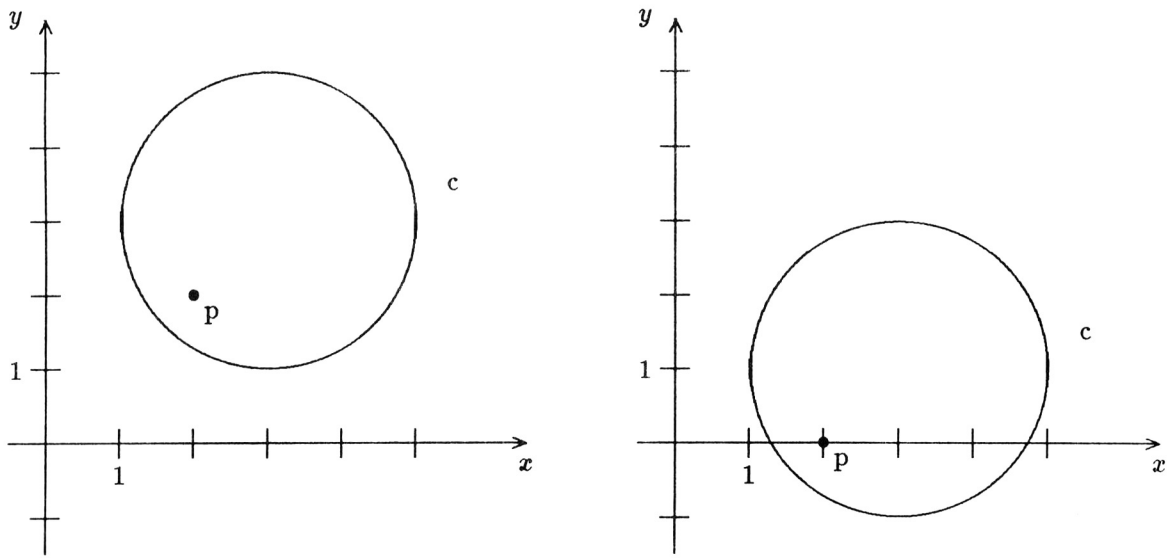
1.1 Overview

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.

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

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



before moving

after moving

Figure 1: Points and circles in the plane

is derived from an example by [CP89]: it is about points and circles with Cartesian coordinates in the plane. Points and circles can be moved in the plane 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 to zero. There are two methods for inspecting the instance variables because they are not directly visible from the outside. The method *move* changes the position of the receiver. In object oriented terminology the O'small expression *p.m(a)* stands for the sending of m with argument a to the receiver p. There is a method for the distance from the origin and a method that returns TRUE if the receiver is closer to the origin than the argument. Booleans, numbers, and some standard functions on them are primitive.

The class *Circle*, which inherits instance variables and methods from *Point*, has an additional instance variable for the radius, methods for reading and changing the radius, and it redefines *distFromOrg*. For the redefinition of *distFromOrg* the *distFromOrg*-definition of the superclass is referred to by *super.distFromOrg*. Note that the inherited function *closerToOrg* has not been redefined in the class *Circle*. Nevertheless with *self.distFromOrg* in the body of *closerToOrg* the *distFromOrg*-definition 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 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.

```
class Point subclassOf Base
def var xComp := 0; var yComp := 0
in meth x() xComp
   meth y() yComp
   meth move(X,Y) xComp := X+xComp; yComp := Y+yComp
   meth distFromOrg()  sqrt(xComp*xComp + yComp*yComp)
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg
ni

class Circle subclassOf Point
def var radius := 0
in meth r() radius
   meth setR(r) radius := r
   meth distFromOrg() max(0, super.distFromOrg - radius)
ni

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);

   output p.closerToOrg(c)
ni
```

Figure 2: Example program in O'small

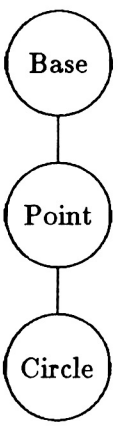


Figure 3: Inheritance graph

Definition 1 A *record* is a finite mapping from a set of labels onto a set of values. A record is denoted by $\begin{bmatrix} x_1 \mapsto v_1 \\ \vdots \quad \mapsto \vdots \\ x_n \mapsto v_n \end{bmatrix}$ with labels x_i and values v_i. All labels that are not in the list are mapped onto $\perp$. The empty record, where all labels are mapped onto $\perp$ is denoted by $[\,]$.

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

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

$\oplus$ is left-associative.

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

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

Let $\mathbf{Y}$ denote 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 $FACT$:

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

A *class* is a generator that creates objects. The domain of classes is Class = Object $\rightarrow$ Object. We use the term domain instead of **type** because of the following chapters. C is an example of a class:

$$C = \lambda s.\begin{bmatrix} sqr \mapsto \lambda x.x * x \\ abs \mapsto \lambda x.\sqrt{s.sqr(x)} \end{bmatrix}$$

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

$$c = \mathbf{Y}(C) = \begin{bmatrix} sqr \mapsto \lambda x.x * x \\ abs \mapsto \lambda x.\sqrt{\mathbf{Y}(C).sqr(x)} \end{bmatrix} = \begin{bmatrix} sqr \mapsto \lambda x.x * x \\ abs \mapsto \lambda x.\sqrt{x * x} \end{bmatrix}$$

Inheritance is the derivation of a new generator from an existing one, 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. From now on we will consider wrappers that operate on classes. Thus the domain of wrappers is Wrapper = Object $\rightarrow$ Object $\rightarrow$ Object.

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:

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

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

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

where $w \triangleright c = (w \cdot c) \oplus c = w(c) \oplus c$ 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. *Base* is the class whose objects are empty records. The semantics of a class definition is a wrapper being wrapped around the superclass, and this results in a new class. As pointed out above we can get objects by applying the fixed point operator to a class. Examples of wrapper applications can be found in [Coo89].

Whereas here objects have no state, the definitions presented so far will be used in section 4 where objects do have 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. We chose SMALL and Gordon's notation for direct semantics because it is fairly standard. O'small is limited in general: it does not include recursive classes, 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 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. 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 classes. 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 newly introduced semantic domains are *Object*, *Class*, and *Wrapper*. Objects, classes, and wrappers were introduced in section 3. Their domains were:

$$\begin{array}{lcl} \text{Object} & = & \text{Record} \\ \text{Class} & = & \text{Object} \rightarrow \text{Object} \\ \text{Wrapper} & = & \text{Object} \rightarrow \text{Object} \rightarrow \text{Object} \end{array}$$

These domains have to be modified to include state. *Object* remains unchanged, except that the record values are different now because the state is hidden inside them. The domain of wrappers is completely determined by the domain of classes because wrappers take fixed points of classes and return classes.

The domain of classes

To understand the semantic domain of classes 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 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.

For the domain of classes there are two possible choices. The problem with the introduction of state is as follows. The method environment is recursive and a fixed point operation has to be applied. The allocation of instance variables must not be recursive – otherwise repeated allocation in the store may be the result. One choice consists of feeding the current store before applying the fixed point operation. This results in

$$\text{Store} \rightarrow [(\text{Object} \rightarrow \text{Object}) \times \text{Store}]$$

as the domain for classes. The store has to appear in the domain and in the codomain because the instance variables of objects have to be allocated. (Object $\rightarrow$ Object) appears in the codomain because the fixed point operation has to be applied. The domain looks simple but the semantic clauses for object creation and class definition become cluttered.

We decided to keep the clutter in these clauses to a minimum and therefore opt for the second choice: the store is fed after the fixed point operation. For the fixed point operator to be applied to it the domain of the class must be

$$\alpha \rightarrow \alpha$$

where α is any domain (the domain of classes was $Object \rightarrow Object$ in section 3). A function is needed for the allocation of all instance variables of the new object. They include the instance variables declared in superclasses. 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 is:

$$\text{Store} \rightarrow [\text{Object} \times \text{Store}]$$

This is our α. Thus the *domain for classes* is:

$$(\text{Store} \to [\text{Object} \times \text{Store}]) \to (\text{Store} \to [\text{Object} \times \text{Store}])$$

This type is more complicated than the type of the other solution.

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

Compound syntactic domains

Pro	the domain of programs	P
Exp	the domain of expressions	E
CExp	the domain of compound expressions	C
Var	the domain of variable declarations	V
Cla	the domain of class declarations	K
Meth	the domain of method declarations	M

Method declarations are distinguished from variable and class declarations because methods are declared in classes only. In lieu of commands [Gor79] we have compound expressions. Their syntactic appearance is similar to commands but compound expressions return a value, whence the name.

4.4.2 Syntactic Clauses

P ::= K C

K ::= class I_1 subclassOf I_2 def V in M | K_1 K_2 | ϵ

C ::= E | I := E | output E | if E then C_1 else C_2 | while E do C | def V in C | C_1;C_2

E ::= B | true | false | read | I | $I_1.I_2(E_1,\ldots,E_n)$ | new E | E_1 O E_2

V ::= var I := E | V_1 V_2 | ϵ

M ::= meth $I(I_1,\ldots,I_n)$ C | M_1 M_2 | ϵ

Class, variable, and method declarations may be empty.

4.5 Semantics of O'small

4.5.1 Semantic Domains

Primitive semantic domains:

Unit	the one-point-domain	u
Bool	the domain of booleans	b
Loc	the domain of locations	i
Bv	the domain of basic values	e

The element of *Unit* is denoted by ***unit***. Compound semantic domains are defined by the following domain equations:

$\text{Record}_{\alpha,\beta}$	$= \alpha \to [\beta + \{\bot\}]$	records	
Env	$= \text{Record}_{Ide,Dv}$	environments	r
Object	$= \text{Record}_{Ide,Dv}$	objects	o
Dv	$= \text{Loc} + \text{Rv} + \text{Method}_n + \text{Class}$	denotable values	d
Sv	$= \text{File} + \text{Rv}$	storable values	v
Rv	$= \text{Unit} + \text{Bool} + \text{Bv} + \text{Object}$	R-values	e
File	$= \text{Rv}^*$	files	i
Store	$= \text{Record}_{Loc,Sv}$	stores	s
Method_n	$= \text{Dv}^n \to \text{Store} \to [\text{Dv} \times \text{Store}]$	method values	m
Class	$= \text{Fixed} \to \text{Fixed}$	class values	c
Fixed	$= \text{Store} \to [\text{Object} \times \text{Store}]$	fixed values	x
Wrapper	$= \text{Fixed} \to \text{Class}$	wrapper values	w
Ans	$= \text{File} \times \{\text{error, stop}\}$	program answers	a

Records are polymorphic. Domains Method_n are needed for each $n \in \mathbb{N}_0$. Fixed values can create objects but are not suited for inheritance. They are the results of fixed point operations applied to classes.

4.5.2 Semantic Clauses

The following semantic functions are primitive:

$\mathsf{B}: \text{Bas} \to \text{Bv}$

$\mathsf{O}: \text{BinOp} \to \text{Rv} \to \text{Rv} \to \text{Store} \to [\text{Dv} \times \text{Store}]$

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 leaves the store unchanged. The remaining semantic functions will be defined by clauses and have the following types:

$$
\begin{array}{ll}
\mathsf{P} & : \mathrm{Pro} \to \mathrm{File} \to \mathrm{Ans} \\
\mathsf{R},\ \mathsf{E} & : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Store} \to [\mathrm{Dv} \times \mathrm{Store}] \\
\mathsf{C} & : \mathrm{CExp} \to \mathrm{Env} \to \mathrm{Store} \to [\mathrm{Dv} \times \mathrm{Store}] \\
\mathsf{V} & : \mathrm{Var} \to \mathrm{Env} \to \mathrm{Store} \to [\mathrm{Env} \times \mathrm{Store}] \\
\mathsf{K} & : \mathrm{Cla} \to \mathrm{Env} \to \mathrm{Store} \to [\mathrm{Env} \times \mathrm{Store}] \\
\mathsf{M} & : \mathrm{Meth} \to \mathrm{Env} \to \mathrm{Env}
\end{array}
$$

Definitions

Differing from [Gor79] we use record notation for environments and stores. Alternatives are denoted in braces. Note that in the following clause *err*, *inp* and *out* are locations and not identifiers.

Note that the dot of λ-abstractions is the operator with the least precedence. An abstracted variable is bound until the end of the clause. This is only in a few cases indicated by extra parentheses.

$$
\begin{array}{l}
\mathsf{P}[\![\mathrm{K\ C}]\!]\mathrm{i} = \text{extractans } \mathrm{s}_{final} \\
\quad \text{where} \\
\quad \text{extractans} = \lambda \mathrm{s}.(\mathrm{s\ out}, \left\{ \begin{array}{l} \text{error, if s err} \\ \text{stop , otherwise} \end{array} \right\}) \\
\quad (\mathrm{r}_{class}, \_) = \mathsf{K}[\![\mathrm{K}]\!]\mathrm{r}_{initial}\ \mathrm{s}_{initial} \\
\quad (\_, \mathrm{s}_{final}) = \mathsf{C}[\![\mathrm{C}]\!]\mathrm{r}_{class}\ \mathrm{s}_{initial} \\
\quad \mathrm{r}_{initial} = \left[\ \mathrm{Base} \mapsto \lambda \mathrm{o}.\lambda \mathrm{s}.\text{result } [\,]\ \right]
\end{array}
$$

$$
\mathrm{s}_{initial} = \left[\begin{array}{l} \mathrm{err} \mapsto \mathrm{false} \\ \mathrm{inp} \mapsto \mathrm{i} \\ \mathrm{out} \mapsto \epsilon \end{array} \right]
$$

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 the "empty" class *Base*. The initial environment is enriched by the declared classes. Then the compound expression is evaluated 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.

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

The semantic function R produces R-values.

$$
\begin{array}{lcl}
\mathsf{E}[\![\mathrm{B}]\!]\mathrm{r} & = & \mathrm{result}(\mathsf{B}[\![\mathrm{B}]\!]) \\
\mathsf{E}[\![\mathrm{true}]\!]\mathrm{r} & = & \mathrm{result\ true} \\
\mathsf{E}[\![\mathrm{false}]\!]\mathrm{r} & = & \mathrm{result\ false} \\
\mathsf{E}[\![\mathrm{read}]\!]\mathrm{r} & = & \mathrm{cont\ inp} \star \lambda \mathrm{i}.\lambda \mathrm{s}.\left\{\begin{array}{l} \mathrm{seterr\ s} \qquad\qquad\qquad\quad , \text{if } \mathrm{i} = \epsilon \\ (\mathrm{hd\ i}, [\mathrm{inp} \mapsto \mathrm{tl\ i}] \oplus \mathrm{s}), \text{otherwise} \end{array}\right\} \\
\mathsf{E}[\![\mathrm{I}]\!]\mathrm{r} & = & \mathrm{result\ (r\ I)} \star \mathrm{Dv?} \\
\mathsf{E}[\![\mathrm{I_1.I_2(E_1,\ldots,E_n)}]\!]\mathrm{r} & = & \mathsf{R}[\![\mathrm{I_1}]\!]\mathrm{r} \star \mathrm{Object?} \star \lambda \mathrm{o}.(\mathrm{result(o\ I_2)} \star \mathrm{Method?} \star \\
 & & \lambda \mathrm{m}.\mathsf{R}[\![\mathrm{E_1}]\!]\mathrm{r} \star \lambda \mathrm{d_1}.\ \ldots \mathsf{R}[\![\mathrm{E_n}]\!]\mathrm{r} \star \lambda \mathrm{d_n}.\mathrm{m(d_1,\ldots,d_n)})
\end{array}
$$

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 message that is then applied to the parameters.

$$
\mathsf{E}[\![\mathrm{new\ E}]\!]\mathrm{r} \quad = \mathsf{E}[\![\mathrm{E}]\!]\mathrm{r} \star \mathrm{Class?} \star \lambda \mathrm{c}.\lambda \mathrm{s}.(\mathbf{Y}\ \mathrm{c})\mathrm{s}
$$

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

$$
\mathsf{E}\ [\![\mathrm{E_1\ O\ E_2}]\!]\mathrm{r} \quad = \quad \mathsf{R}[\![\mathrm{E_1}]\!]\mathrm{r} \star \lambda \mathrm{e_1}.\mathsf{R}[\![\mathrm{E_2}]\!]\mathrm{r} \star \lambda \mathrm{e_2}.\mathsf{O}[\![\mathrm{O}]\!](\mathrm{e_1}, \mathrm{e_2})
$$

$$
\begin{array}{lcl}
\mathsf{C}[\![\mathrm{E}]\!]\mathrm{r} & = & \mathsf{E}[\![\mathrm{E}]\!]\mathrm{r} \\
\mathsf{C}[\![\mathrm{I := E}]\!]\mathrm{r} & = & \mathsf{E}[\![\mathrm{I}]\!]\mathrm{r} \star \mathrm{Loc?} \star \lambda \mathrm{l}.\ \mathsf{R}[\![\mathrm{E}]\!]\mathrm{r} \star (\mathrm{update\ l}) \\
\mathsf{C}[\![\mathrm{output\ E}]\!]\mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\mathrm{r} \star \lambda \mathrm{e}.\lambda \mathrm{s}.(\mathrm{unit}, [\mathrm{out} \mapsto \mathrm{append(s\ out,e)}] \oplus \mathrm{s}) \\
\mathsf{C}[\![\mathrm{if\ E\ then\ C_1\ else\ C_2}]\!]\mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\mathrm{r} \star \mathrm{Bool?} \star \mathrm{cond}(\mathsf{C}[\![\mathrm{C_1}]\!]\mathrm{r}, \mathsf{C}[\![\mathrm{C_2}]\!]\mathrm{r}) \\
\mathsf{C}[\![\mathrm{while\ E\ do\ C}]\!]\mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\mathrm{r} \star \mathrm{Bool?} \star \\
 & & \mathrm{cond}(\mathsf{C}[\![\mathrm{C}]\!]\mathrm{r} \star \lambda \mathrm{e}.\mathsf{C}[\![\mathrm{while\ E\ do\ C}]\!]\mathrm{r}, \mathrm{result\ unit}) \\
\mathsf{C}[\![\mathrm{def\ V\ in\ C\ end}]\!]\mathrm{r} & = & \mathsf{V}[\![\mathrm{V}]\!]\mathrm{r} \star \lambda \mathrm{r'}.\mathsf{C}[\![\mathrm{C}]\!](\mathrm{r'} \oplus \mathrm{r}) \\
\mathsf{C}[\![\mathrm{C_1;\ C_2}]\!]\mathrm{r} & = & \mathsf{C}[\![\mathrm{C_1}]\!]\mathrm{r} \star \lambda \mathrm{e}.\mathsf{C}[\![\mathrm{C_2}]\!]\mathrm{r}
\end{array}
$$

The result of assignment, output-term, and while-loop is *unit*. In the sequence the transmitted value is discarded. This practice has been adopted from ML [Mil84].

$$
\begin{array}{l}
\mathsf{K}[\![\mathrm{I_1\ subclassOf\ I_2\ def\ V\ in\ M}]\!]\mathrm{r} \\
\quad = \mathsf{E}[\![\mathrm{I_2}]\!]\mathrm{r} \star \mathrm{Class?} \star \lambda \mathrm{g}.\mathrm{result}[\mathrm{I_1} \mapsto \mathrm{w}\ \boxed{\triangleright}\ \mathrm{c}] \\
\quad \text{where} \\
\quad \mathrm{w} = \lambda \mathrm{x}_{self}.\lambda \mathrm{x}_{super}.\lambda \mathrm{s}_{create}.\ \left(\left[\begin{array}{ll} \mathrm{self} & \mapsto \mathrm{r}_{self} \\ \mathrm{super} & \mapsto \mathrm{r}_{super} \end{array}\right] \oplus \mathrm{r}_{local} \oplus (\mathsf{M}[\![\mathrm{M}]\!]\mathrm{r})\ ,\ \mathrm{s}_{new}\right) \\
\qquad \text{where} \\
\qquad (\mathrm{r}_{super}, \mathrm{s}_{super}) = \mathrm{x}_{super}\ \mathrm{s}_{create} \\
\qquad (\mathrm{r}_{local}, \mathrm{s}_{new}) = \mathsf{V}[\![\mathrm{V}]\!]\mathrm{r}\ \mathrm{s}_{super} \\
\qquad (\mathrm{r}_{self}, \_) = \mathrm{x}_{self}\ \mathrm{s}_{create}
\end{array}
$$

The result of the evaluation of a class declaration is the binding of a class to the class name.

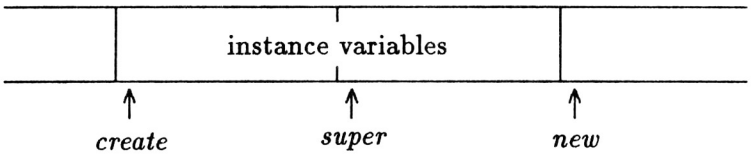


Figure 4: The store during object creation

The store remains unchanged when a class is declared. The wrapper w takes a "fixed" for self reference, a "fixed" for reference to the superclass, and a store as parameters. The store parameter is fed at object creation time, x_{self} is fed at the fixed point operation, and x_{super} is fed at the wrapper application. The wrapper evaluates the method 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 and not in any subclass. Thus we have encapsulated instance variables.

Let us consider what happens at object creation time during the evaluation of the inner where-clause above. Figure 4 shows the store with arrows pointing at the first free cell of the store with the respective index. x_{super} is applied to the store at object creation time. This results in the method environment of the superclass; but also the part of the instance variables defined in the superclass are allocated and the first free cell of the store is indicated by *super* in the figure. The new instance variables for the current class are declared in V. V has to be evaluated in s_{super} to put the new instance variables "behind" the inherited ones. Of course it could have been done the other way round. All instance variables are allocated now and the resulting store (s_{new}) is passed on to the remaining program. There is however a third line where x_{self} is applied to s_{create}. x_{self} is the recursive part and r_{self} is the resulting recursive environment. x_{self} has to be applied to s_{create} because its instance variables are the ones that have just been allocated.

The careful reader may have noticed already that the resulting store is not needed. This is indicated by an underscore. 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.

$$
\begin{aligned}
&\mathsf{K}[\![\mathrm{K_1\ K_2}]\!]\mathrm{r} = \mathsf{K}[\![\mathrm{K_1}]\!]\mathrm{r} \star \lambda \mathrm{r'}.\ \mathrm{r'} \oplus (\mathsf{K}[\![\mathrm{K_2}]\!]\mathrm{r}) \\
&\mathsf{K}[\![\epsilon]\!]\mathrm{r} = \mathrm{result}\ [\,] \\
&\mathsf{V}[\![\mathrm{var\ I} := \mathrm{E}]\!]\mathrm{r} = \mathsf{R}[\![\mathrm{E}]\!]\mathrm{r} \star \lambda \mathrm{d}.\ \mathrm{new} \star \lambda \mathrm{l}.\lambda \mathrm{s}.\ ([\mathrm{I} \mapsto \mathrm{l}], [\mathrm{l} \mapsto \mathrm{d}] \oplus \mathrm{s}) \\
&\mathsf{V}[\![\mathrm{V_1\ V_2}]\!]\mathrm{r} = \mathsf{V}[\![\mathrm{V_1}]\!]\mathrm{r} \star \lambda \mathrm{r'}.\ \mathrm{r'} \oplus (\mathsf{V}[\![\mathrm{V_2}]\!]\mathrm{r}) \\
&\mathsf{V}[\![\epsilon]\!]\mathrm{r} = \mathrm{result}\ [\,] \\
&\mathsf{M}[\![\mathrm{meth\ I}(\mathrm{I_1},\ldots,\mathrm{I_n})\ \mathrm{C}]\!]\mathrm{r} = \left[RMI \mapsto \lambda \mathrm{d_1}.\ \ldots . \lambda \mathrm{d_n}.\ \mathsf{C}[\![\mathrm{C}]\!]\left(\begin{bmatrix} \mathrm{I_1} \mapsto \mathrm{d_1} \\ \vdots \mapsto \vdots \\ \mathrm{I_n} \mapsto \mathrm{d_n} \end{bmatrix} \oplus \mathrm{r} \right) \right] \\
&\mathsf{M}[\![\mathrm{M_1\ M_2}]\!]\mathrm{r} = (\mathsf{M}[\![\mathrm{M_2}]\!]\mathrm{r}) \oplus (\mathsf{M}[\![\mathrm{M_1}]\!]\mathrm{r}) \\
&\mathsf{M}[\![\epsilon]\!]\mathrm{r} = [\,]
\end{aligned}
$$

Method definitions are not recursive. Recursion and the calling of other methods is 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] needs an additional level of inheritance 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. An additional level of inheritance can be easily introduced into the semantics. But on the contrary of what might be expected, an additional level of inheritance is superfluous in O'small. It is sufficient to bind *current* to $\lambda x.x_{self}$. The class definition clause is thus changed to:

$$
\begin{array}{l}
\mathsf{K}[\![\mathrm{I}_1 \text{ subclassOf } \mathrm{I}_2 \text{ def V in M}]\!]\mathrm{r} \\
\quad = \mathsf{E}[\![\mathrm{I}_2]\!]\mathrm{r} \ \star \ \text{Class?} \ \star \ \lambda \mathrm{g.result}[\mathrm{I}_1 \mapsto \mathrm{w} \ \boxed{\triangleright} \ \mathrm{c}] \\
\quad \text{where} \\
\quad \mathrm{w} = \lambda \mathrm{x}_{self}.\lambda \mathrm{x}_{super}.\lambda \mathrm{s}_{create}.\ \left(\begin{bmatrix} \text{self} & \mapsto \mathrm{r}_{self} \\ \text{current} & \mapsto \lambda \mathrm{x.x}_{self} \\ \text{super} & \mapsto \mathrm{r}_{super} \end{bmatrix} \oplus \mathrm{r}_{local} \oplus (\mathsf{M}[\![\mathrm{M}]\!]\mathrm{r}) \ , \ \mathrm{s}_{new} \right) \\
\qquad \text{where } \ldots
\end{array}
$$

The abstraction of x in the term bound to *current* makes this term a member of the domain of classes. The fixed point of this term is of course x_{self}.

5 Conclusion

We have started with a semantics of inheritance as presented in [CP89]. This semantics does not include 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 hope to have answered this question in the affirmative. For the description of objects with state the domains of classes and wrappers had to be adjusted accordingly. The introduction of state was possible without making the central semantic clauses of class declaration and object creation complicated, at least at the surface; the complications are limited to the inner where clause of class declaration. The creation of objects of the current class is possible without an extra level of inheritance. O'small is executable because the semantics has been translated into 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, Christian Neusius, and Mario Wolczko for commenting on drafts of my paper. I am grateful to Andreas Gündel, Carl Gunter, 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. revised in: Information and Computation, Vol. 76, 1988, pp. 138-164.

[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.

[Mil84] Robin Milner. A proposal for standard ML. In *Symposium on Lisp and Functional Programming*, pages 184–197, ACM, Austin Texas, 1984.

[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:

$$f : \left\langle \begin{array}{c} Store \\ D_1 \to Store \end{array} \right\rangle \to [D_2 \times Store], \quad g : D_2 \to Store \to [D_3 \times Store]$$

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 you choose the upper alternative, you have to choose the upper alternative in every brace below. If above in the braces you choose the lower alternative, you have to choose the lower alternative in every brace below. Then the composition of f and g has type

$$f \star g : \left\langle \begin{array}{c} Store \\ D_1 \to Store \end{array} \right\rangle_a \to [D_3 \times Store]$$

and is defined by

$$f \star g = \left\langle \begin{array}{c} \lambda s_1 \\ \lambda d_1.\lambda s_1 \end{array} \right\rangle \cdot \begin{cases} (\bot, s_2), \text{ if } s_2 \ err \\ g\ d_2\ s_2, \text{ otherwise} \end{cases} \quad \text{where} \quad (d_2, s_2) = \left\langle \begin{array}{c} f s_1 \\ f d_1 s_1 \end{array} \right\rangle$$

$\star$ is left associative. The definition of $\triangleright$ in section 3 is based on the left-preferential combination of records (denoted by $\oplus$). This symbol is also overloaded in the semantic equations. If the arguments of $\oplus$ are of the domain *Fixed* then $\oplus$ stands for:

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

where $\oplus_{lpr}$ stands for the operation on records that 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 + {⊥}] × Store] ..... Contents of a location
cnt = λl.λs.(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. { result d, if isD d
         { seterr , otherwise

deref : Dv → Store → [Dv × Store] ..... Dereferencing

deref = λe. { cont e , if isLoc e
            { result e, otherwise

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

If new s = (l,s) then s l = ⊥ 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.(⊥, [err ↦ true] ⊕ s)

update : Loc → Dv → Store → [Dv×Store] ..... Updating of a location
update l = Sv? ⋆ λe.λs.(unit, [l ↦ e] ⊕ s)