Formal Aspects of Computing (1993) 5: 181–207

Formal Aspects of Computing

Denotational Semantics of an Object-Oriented Programming Language with Explicit Wrappers

Andreas V. Hense

Lehrstuhl für Programmiersprachen und Übersetzerbau, Universität des Saarlandes, Saarbrücken, Germany

Keywords: Class inheritance; Denotational semantics; Hierarchy inheritance; Multiple inheritance; Wrapper

Abstract. Object-oriented languages have traditionally been described by method-lookup-semantics. Their denotational semantics have appeared and matured only recently. Cook's wrapper semantics without state shows the essence of inheritance much clearer than method-lookup-semantics.

In this article, we show how wrapper semantics can describe an object-oriented language *with state* while keeping its original clear structure. We then extend our object-oriented language by so called *explicit wrappers*. Wrappers that are used for the description of the semantics of an "ordinary" object-oriented language emerge from the semantics level and are included into the language itself. This unusual step is being justified by a greater reusability of code. With explicit wrappers and single inheritance, one variety of multiple inheritance can be expressed.

1. Introduction

One of the main advantages of object-oriented programming is an increased reusability of code. This is the reason for its relatively early spreading into industrial contexts, while the foundational research is still very active. *Class-inheritance* contributes to structured code reuse. The terminology of object-oriented programming suggests that *classes* are types, *subclasses* are subtypes, and so forth. This has lead to confusion in the past. However, class-inheritance is *not* subtyping [CHC90]

Correspondence and offprint requests to: Andreas V. Hense, Lehrstuhl für Programmiersprachen und Übersetzerbau, FB–14 Informatik, Universität des Saarlandes, 6600 Saarbrücken 11, Germany, e-mail: hense@cs.uni-sb.de

but an intricate mechanism, featuring dynamic binding together with some clever naming conventions (the pseudo-variables of SMALLTALK [GoR89]). Inheritance is a mechanism for incremental modification. In our framework, it is possible to redefine methods in such a way that their semantics in the subclass has nothing to do with their semantics in the superclass. As long as certain minimal requirements on type compatibility are guaranteed [Coo89], no errors will occur. However, for a disciplined programming style, we require more, and advocate a (disciplined) version of inheritance allowing certain compatibility assumptions on subclasses. Otherwise, methods will be inherited just because they happen to fit into the current scheme and many dependencies between classes will hinder modifications in implementations. The following classification of incremental modifications is adapted from [WeZ88]:

- *Behaviour-compatible modification*: The entries to be modified are specified (e.g. by many sorted algebras [EhM85]). Syntax is specified by signatures and semantics by axioms. Modifications are behaviour-compatible subalgebras.
- *Signature-compatible modification*: Like above but without semantics specifications. Subsignatures are in general not behaviour-compatible.
- *Name-compatible modification*: Modifications have a superset of the labels.
- *Inheritance with cancellation*:: Traditional inheritance focuses on subtypes defined by increasing the severity of constraints. Cancellation relaxes constraints (e.g., in a subclass, a method can be dropped).

Cancellation may occur at the level of behaviour, signatures, or names. By construction, O'SMALL is at the level of name-compatible modification but cancellation should be avoided by the programmer at the signature level and "somewhere below" the behaviour level. Here, "below" means that we want compatibility (substitutability [WeZ88]). The issue of behavioural compatibility has been approached pragmatically in Eiffel [Mey88], with Hoare-logic restrictions being checked at run time, but a solution to the whole problem is still a subject of research [ESS89, Gün90]. Excluding cancellation at the signature level can be achieved by static type checking [Hen93].

This article's main concern is language design. A closer look at the inheritance mechanism with denotational semantics using wrappers [Coo89] has lead us to the idea of *explicit wrappers*.[1] A wrapper is at first an element of the semantics describing the incremental modification of a subclass definition. Making wrappers explicit means adding a new feature to the language. The programmer can use his modifications more flexibly, and thus the reusability of code is even higher than in "ordinary" object-oriented programming languages.

We show examples of increased code reusability thanks to explicit wrappers. Furthermore, explicit wrappers allow us to see multiple inheritance from a new angle: with explicit wrappers and single inheritance we are able to model certain cases of multiple inheritance.

1.1. Overview

Section 2 describes the semantic domains needed for object-oriented languages and class inheritance in a purely functional framework. The internal state of ob-

[1] First appeared in [Hen90] and developed independently in [BrC90].

jects has been abstracted. The definitions of classes and wrappers can thus be kept simple. To make this article self-contained, section 3 gives the the denotational semantics of an object-oriented language with state called O'SMALL [Hen91]. Readers who just want an intuitive understanding may skip section 3. Section 4 is the main section of this article. The semantic construct that leads to a clear description of class inheritance in sections 2 and 3 is now being used in the language itself. One of the consequences is that multiple inheritance may become less necessary because one of its varieties can now be modeled by single inheritance. O'SMALL, the language in the examples, appears in several "dialects":

- functional O'SMALL (section 2)
- (classical) O'SMALL (section 3)
- O'SMALL with explicit wrappers (section 4)

Functional O'SMALL is introduced informally and differs considerably from the other dialects. Classical O'SMALL is a full object-oriented language with a denotational semantics. O'SMALL with explicit wrappers is a slight extension of classical O'SMALL. The differences are presented in section B.

2. Domain Theory and Object-Oriented Languages

The semantics of O'SMALL itself will be contained in section 3. This section describes the semantic domains we use for an object-oriented language. To simplify things, we consider functional O'SMALL, i.e. we abstract from state. Programs of functional O'SMALL contain no assignments. In functional O'SMALL classes have parameters, as opposed to classical O'SMALL. This approach is based on the work of Cook [Coo89]. We will make a few remarks on fixed point semantics and its appearance in the description of object-oriented languages, followed by some basic definitions on records. Records model objects. These basic definitions will also be used in later sections. After these preliminaries, the ground will be prepared for the definition of the semantic domains and the inheritance mechanism in functional O'SMALL.

Self-reference and application of functions to themselves pose mathematical problems. Yet, recursive procedures or functions are common in programming, and we will also need them in the remainder of this article. Scott [Sco76] provided a basis for mathematical structures, called complete partial orders or cpo's for short, that are suited for the description of recursive programs. For an overview refer to [Bar81, HiS86]. Let us give an informal example of a recursive definition in an O'SMALL like notation:

```
meth fac(n)    if n=0 then 1 else n * fac(n-1)
```

This definition is recursive or self-referential. In SMALLTALK and O'SMALL, self-reference is standardized syntactically by the pseudo-variable `self`. Instead of just applying a function to an argument, object-oriented languages send a message to an object. *Message sending* is record selection. For the above example, we thus obtain the O'SMALL program fragment:

```
meth fact(n)    if n=0 then 1 else n * self.fact(n-1)
```

This is a method definition that may appear in a class. In order to access this method, we have to send a message with selector `fact` to an object of this class,

but, because the definition is self-referential, one does still not know what it denotes. We transform the above definition into non-recursive form by explicitly abstracting the self-reference. A λ-calculus-like meta language is used for semantic considerations.

$$Fact = \lambda s.[fact \mapsto \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * s.fact(n-1)]$$

Fact maps records to records. Its definition is not recursive. The abstracted variable s plays the role of `self`. The fixed-point theorem [Tar55] guarantees the existence of a least fixed-point for all continuous functions from a cpo to itself.[2] All functions considered here are continuous. Let F be such a function. Then we write FIX(F) for the least fixed-point of F. If $f = \text{FIX}(F)$ then $F(f) = f$. The function *fac*, we intended to define in the first place, is now the fact-component of *Fact*'s fixed-point:

$$fac = \text{FIX}(Fact).fact$$

2.1. Records

The following basic definitions will be used in this section as well as in the semantics definition of O'SMALL (section 3). Records are needed for the modeling of objects.

Definition 2.1. A *record* is a finite mapping from a set of labels to a set of values. A record is denoted by $\begin{bmatrix} x_1 & \mapsto & v_1 \\ & \vdots & \\ x_n & \mapsto & v_n \end{bmatrix}$ with labels x_i and values v_i. All labels which are not in the list are mapped to $\perp$. The empty record, where all labels are mapped to $\perp$, is denoted by [ ]. Selection of a component x in record r is denoted by $r.x$.

Definition 2.2. Let $dom(r) = \{x \mid r(x) \neq \perp\}$. The *left-preferential concatenation of records* is defined by

$$(r \oplus s)(x) = \begin{cases} r(x) & \text{if } x \in dom(r), \\ s(x) & \text{if } x \in dom(s) - dom(r), \\ \perp & \text{otherwise.} \end{cases}$$

The idea of $\_ \oplus \_$ is the composition of two records where the left one wins in case of conflicts. The following function defines modification of records using left-preferential concatenation of records.

Definition 2.3. $\_ \triangleright \_$: (Record $\rightarrow$ Record) $\rightarrow$ Record $\rightarrow$ Record
takes a function on records and a record, and yields a record. It is defined by

$$\triangleright(f)(b) = f(b) \oplus b.$$

The idea of $\_ \triangleright \_$ is that a function uses and then overwrites a record. The following definition will be explained later. Here we restrict ourselves to saying that together with a wrapper (see below), the modification function for records

[2] The original theorem is formulated for complete lattices.

can be transformed to a modification function for classes (see below):

$$(\boxed{\triangleright}\ w) : \text{Class} \to \text{Class}$$

Note that we freely mix one-place or two-place writing of functions (*currying*) and prefix or infix notation. When written as an infix operator, $\_\ \boxed{\triangleright}\ \_$ is right-associative.

Definition 2.4. We define a higher-order function $\_\ \boxed{\_}\ \_ : (\alpha \to \beta \to \gamma) \to (\delta \to \alpha) \to (\delta \to \beta) \to \delta \to \gamma$ that makes of a binary operator $\_ * \_ : \alpha \to \beta \to \gamma$ its *self-distributing version* denoted by $\_\ \boxed{*}\ \_ : (\delta \to \alpha) \to (\delta \to \beta) \to \delta \to \gamma$ and defined by $a\ \boxed{*}\ b = \lambda s.(a\ s) * (b\ s)$.

2.2. Inheritance

Inheritance is a way of class modification. For its description we will present the notions of object, class, wrapper, and an inheritance function. For the reader's convenience Fig. 1 summarizes domains and functions.

Class	=	Object → Object	meta variable c
Wrapper	=	Object → Class	meta variable w
$\_ \oplus \_$	:	Object → Class	
$\_ \triangleright \_$	:	Class → Class	
$\boxed{\triangleright}$	:	Wrapper → (Class → Class)	inheritance function
$(\boxed{\triangleright}\ w)$	:	Class → Class	class modification

Fig. 1. Domains and functions.

The object-oriented paradigm consists of objects that communicate by message passing. When an object receives a message it "decides" itself what to do, that is it chooses the method that corresponds to the message selector of the received message. The abstract domain of objects can be represented by a domain of records. Message selection amounts to the selection of a record component. One may want several objects with the same set of methods, and therefore one introduces classes. Classes generate objects. In the general case a class generates similar objects. In this section, for technical reasons, a class generates identical objects. Objects may also send messages to themselves and, thus, classes must provide a means of *self-reference*. We will see that a class generates an object by a fixed-point operation.

When a new class (*subclass*) is defined one wants to refer to an existing class (*superclass*) and say things like: "objects of this class are like objects of the superclass, but this and this are different". In subclass definitions, new methods may be added and existing methods may be redefined. In the redefinition of existing methods, there is a way of referring to the overwritten definition. Together with the inheritance function, wrappers will be able to modify classes. The inheritance function is defined such that the self-references are distributed appropriately. *Fact* is an example of a class with only one method. Let us look at another example of a class. Fig. 2 shows a program in functional O'SMALL whose semantics will be discussed now. We focus on the first class-definition.

```
class Point(a,b) inheritsFrom Base
   meth x() a
   meth y() b
   meth distFromOrg()  sqrt( sqr(self.x)  +  sqr(self.y) )
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg

class Circle(a,b,c) inheritsFrom Point(a,b)
   meth r() c
   meth distFromOrg() max(0, super.distFromOrg - self.r)

let var p = new Point(2,2)
    var c = new Circle(3,3,2)
in . . . ni
```

Fig. 2. Program in functional O'SMALL.

$$Point = \lambda a.\lambda b.\lambda s. \begin{bmatrix} x & \mapsto & a \\ y & \mapsto & b \\ distFromOrg & \mapsto & \sqrt{(s.x)^2 + (s.y)^2} \\ closerToOrg & \mapsto & \lambda p.\ s.distFromOrg < p.distFromOrg \end{bmatrix}$$

Point is the class of points in two dimensional space[3]. An object p of class *Point* is created by:

$$p = \text{FIX}(Point\ 2\ 2) = \begin{bmatrix} x & \mapsto & 2 \\ y & \mapsto & 2 \\ distFromOrg & \mapsto & \sqrt{8} \\ closerToOrg & \mapsto & \lambda p.\ \sqrt{8} < p.distFromOrg \end{bmatrix}$$

On the semantic level, classes are functions of the form $\lambda s.B$ where s is the standardized variable representing self-reference. *Inheritance* is the construction of a new class (the *subclass*) using an existing class (the *superclass*). The formal parameters for self-reference of the superclass and of the subclass "are the same". The additional definitions or modifications in the subclass are modeled by *wrappers*. A wrapper is a function taking two objects as input and returning a new object. The first input object is the parameter for self-reference, like the parameter for self reference of classes. The second input object is the parameter for "super-reference". Wrappers have this special form here, because the language we describe has the pseudo-variables `self` and `super`. In a hypothetical language, if there were further pseudo-variables, wrappers would have more parameters.

Let us now take a look at the second class definition of Fig. 2. Circles are defined as a subclass of the already defined class of points. The wrapper *CIRCLE*[4] contains the differences between points and circles. The variable p is used like the pseudo-variable `super` in SMALLTALK or O'SMALL.

[3] To be precise: (Point a_1 a_2) is a class.
[4] To be precise: (*CIRCLE* a_1 a_2 a_3) is a wrapper.

$$CIRCLE = \lambda a.\lambda b.\lambda c.\lambda s.\lambda p. \begin{bmatrix} r & \mapsto & c \\ distFromOrg & \mapsto & max(0, p.distFromOrg - s.r) \end{bmatrix}$$

The *Circle*-class is created by:

$$Circle = \lambda a.\lambda b.\lambda c.(CIRCLE\ a\ b\ c)\ \boxed{\triangleright}\ (Point\ a\ b)$$

An object c of class *Circle* is created by:

$$c = \text{FIX}(Circle\ 3\ 3\ 2) = \begin{bmatrix} x & \mapsto & 3 \\ y & \mapsto & 3 \\ r & \mapsto & 2 \\ distFromOrg & \mapsto & \sqrt{18} - 2 \\ closerToOrg & \mapsto & \lambda p.\ \sqrt{18} - 2 < p.distFromOrg \end{bmatrix}$$

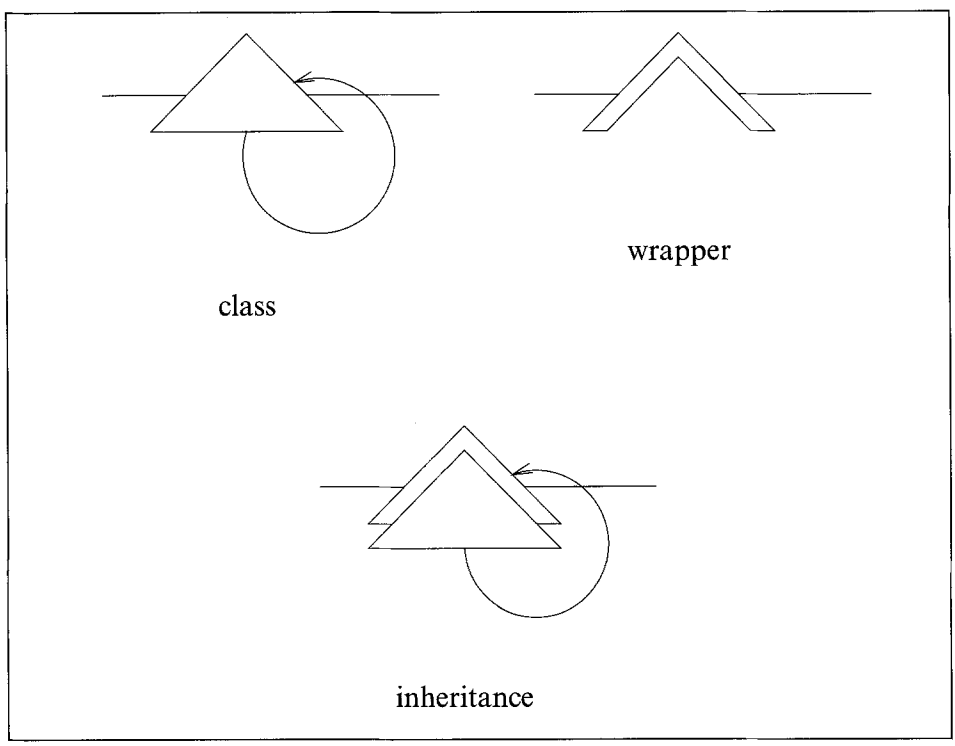

Fig. 3. Iceberg-diagrams.

In order to illustrate the inheritance function, we introduce iceberg-diagrams, an intuitive description of classes and wrappers (Fig. 3). A class is depicted by a triangle, where the visible part (methods) is above the surface and the invisible part (instance variables) below. There is an arrow for self-reference. A wrapper is depicted by an angular shape. Inheritance is a wrapper applied to a class. The

common line of wrappers and classes in the diagram represents the references to `self`, `super`, and the methods that are not redefined. Note that the self-reference of the wrapper and the class now point to the whole.

3. An Object-Oriented Language with State

In this section, we show how to extend the semantics of inheritance without state of section 2 to a semantics of an object-oriented programming language, i.e. we add the state that was abstracted from in section 2. The language being described in this section is the so called classical O'SMALL, which has no explicit wrappers. The semantics of O'SMALL (section 3.2) was formulated around the semantics of the imperative language SMALL [Gor79] but in direct style (without continuations). The aim was to clarify the differences between object-oriented and imperative programming languages. The semantic clauses use some auxiliary functions (section A).

The semantic functions defined in the following are all continuous because they are built by standard constructions (function composition) from continuous functions. Therefore their smallest fixed points exist in cpo's. However, domain theory was not the focus of our attention.

3.1. Syntax of O'SMALL

The syntactic domains are in Fig. 4. Meta variables ranging over domains are listed on the right-hand side. Ide, Bas, and BinOp are primitive, the others compound. 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. The syntactic clauses are in Fig. 5. Class, variable, and method declarations may be empty.

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

Fig. 4. Syntactic domains.

P	::=	K C
K	::=	**class** I_1 **inheritsFrom** 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 | E.I(E_1,...,E_n) | **new** E | E_1 O E_2
V	::=	**var** I := E | V_1 V_2 | ϵ
M	::=	**meth** I(I_1,...,I_n) C | M_1 M_2 | ϵ

Fig. 5. Syntactic clauses.

Unit			one-point-domain	u
Bool			booleans	b
Loc			locations	l
Bv			basic values	
$\text{Record}_{\alpha,\beta}$	=	$\alpha \longrightarrow [\beta + \{\perp\}]$	records	
Env	=	$\text{Record}_{Ide,Dv}$	environments	r
Object	=	$\text{Record}_{Ide,Dv}$	objects	o
Dv	=	Loc + Rv + Method_n + Class	denotable values	d
Sv	=	File + Rv	storable values	v
Rv	=	Unit + Bool + Bv + Object	R-values	e
File	=	Rv^*	files	i
Store	=	$\text{Record}_{Loc,Sv}$	stores	s
Method_n	=	$\text{Dv}^n \longrightarrow$ Store$\longrightarrow$ [Dv×Store]	method values	m
Class	=	Alloc$\longrightarrow$ Alloc	class values	c
Alloc	=	Store$\longrightarrow$ [Object×Store]	allocator values	x
Wrapper	=	Alloc$\longrightarrow$ Class	wrapper values	w
Ans	=	File×{error, stop}	program answers	a

Fig. 6. Semantic domains.

3.2. Semantics of O'SMALL

The semantic domains are in Fig. 6. The first four domains are primitive. *Unit* is the domain needed for the result of compound expressions that do not return a useful value (`while`-expression). Locations are addresses of cells in the store. *Denotable values* can be bound to identifiers in the environment. In O'SMALL, the set of values that can be the result of expressions (*expressible values*) is the same as the set of denotable values. *Storable values* can be put into cells (locations) of the store. Note that one cell can contain a whole file (input and output). *R-values* are the results of evaluating the right-hand sides of assignments. Domains Method_n are needed for each $n \in \mathbf{N}_0$. Allocators can create objects but are not suited for inheritance. They are the results of fixed point operations applied to classes.

B	: Bas⟶ Bv
O	: BinOp⟶ Rv⟶ Rv⟶ Store⟶ [Dv×Store]
P	: Pro⟶ File⟶ Ans
R, E	: Exp⟶ Env⟶ Store⟶ [Dv×Store]
C	: CExp⟶ Env⟶ Store⟶ [Dv×Store]
V	: Var⟶ Env⟶ Store⟶ [Env×Store]
K	: Cla⟶ Env⟶ Store⟶ [Env×Store]
M	: Meth⟶ Env⟶ Env

Fig. 7. Semantic functions.

3.2.1. Semantic Clauses

The types of the semantic functions are in Fig. 7. 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. B and O are primitive. The remaining semantic functions will be defined by clauses.

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, not identifiers. For the definition of auxiliary functions in the following clauses refer to section A.

$$\mathsf{P}[\![\mathsf{K}\ \mathsf{C}]\!]\ \mathrm{i} = \text{extractans}\ \mathrm{s}_{final}$$

where

$$\text{extractans} = \lambda\mathrm{s}.(\mathrm{s}\ \mathrm{out}, \left\{ \begin{array}{l} \text{error, if (s err)} \\ \text{stop , otherwise} \end{array} \right\})$$

$$(\mathrm{r}_{class}, \_) = \mathsf{K}[\![\mathsf{K}]\!]\ \mathrm{r}_{initial}\ \mathrm{s}_{initial}$$

$$(\_, \mathrm{s}_{final}) = \mathsf{C}[\![\mathsf{C}]\!]\ \mathrm{r}_{class}\ \mathrm{s}_{initial}$$

$$\mathrm{r}_{initial} = \left[\begin{array}{lll} \text{Base} & \mapsto & \lambda\mathrm{o}.\lambda\mathrm{s}.\text{result}\ [\,] \end{array} \right]$$

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

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*. It is enriched by the declared classes. Then the compound expression is evaluated. 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*).

Let us look at the example of Fig. 8. In contrast to Fig. 2, objects have an internal state. The move-method, for instance, just changes the internal state of

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

class Circle inheritsFrom Point
def var radius := 0
in meth r()           radius
   meth setR(r)       radius := r
   meth distFromOrg() max(0, super.distFromOrg - self.r)
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);              {results in FALSE}
   p.move(0,-2); c.move(0,-2);
   output p.closerToOrg(c)               {results in FALSE}
ni
```

Fig. 8. Example program in O'SMALL.

a point. The main-program "knows" the class definitions but the store has not been changed by them.

$$
\begin{array}{lcl}
\mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} & = & \mathsf{E}[\![\mathrm{E}]\!]\ \mathrm{r} \star \mathrm{deref} \star \mathrm{Rv?} \\
\mathsf{E}[\![\mathrm{B}]\!]\ \mathrm{r} & = & \mathrm{result}(\mathsf{B}[\![\mathrm{B}]\!]) \\
\mathsf{E}[\![\textbf{true}]\!]\ \mathrm{r} & = & \mathrm{result\ true} \\
\mathsf{E}[\![\textbf{false}]\!]\ \mathrm{r} & = & \mathrm{result\ false} \\
\mathsf{E}[\![\textbf{read}]\!]\ \mathrm{r} & = & \mathrm{cont\ inp} \star \\
 & & \lambda \mathrm{i}.\lambda \mathrm{s}.\left\{ \begin{array}{l} \mathrm{seterr\ s} \qquad\qquad\qquad\quad , \text{if } \mathrm{i} = \epsilon \\ (\mathrm{hd\ i}, [\mathrm{inp} \mapsto \mathrm{tl\ i}] \oplus \mathrm{s}), \text{otherwise} \end{array} \right\} \\
\mathsf{E}[\![\mathrm{I}]\!]\ \mathrm{r} & = & \mathrm{result\ (r\ I)} \star \mathrm{Dv?} \\
\mathsf{E}[\![\mathrm{E.I}(\mathrm{E}_1,\ldots,\mathrm{E}_n)]\!]\ \mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} \star \mathrm{Object?} \star \\
 & & \lambda \mathrm{o}.(\mathrm{result(o\ I)} \star \mathrm{Method?} \star \\
 & & \lambda \mathrm{m}.\ \mathsf{R}[\![\mathrm{E}_1]\!]\ \mathrm{r} \star \lambda \mathrm{d}_1.\ \ldots\ \mathsf{R}[\![\mathrm{E}_n]\!]\ \mathrm{r} \star \lambda \mathrm{d}_n\ . \\
 & & \mathrm{m}(\mathrm{d}_1,\ldots,\mathrm{d}_n))
\end{array}
$$

The semantic function R produces R-values. The read-clause takes an element from the user input and returns it as a result. 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 selector I. This should result in a method that is then applied to the parameters.

The textually first message sending in Fig. 8 is `self.x`.[5] The pseudo variable `self` evaluates to an object. This object is bound to o and with (o I) we get the method. The identifier I is `x` here. The method is bound to m. m applied to all necessary arguments, in this case zero, is the result of this clause.

3.2.2. The New Semantic Domains

Semantic domains for the constructions not appearing in imperative languages must be added. Objects, classes, and wrappers were introduced in section 2. Their domains were:

$$\begin{aligned} \text{Class} &= \text{Object} \longrightarrow \text{Object} \\ \text{Wrapper} &= \text{Object} \longrightarrow (\text{Object} \longrightarrow \text{Object}) \end{aligned}$$

What are the new semantic domains in the semantics of O'SMALL corresponding to the semantics of inheritance? The domains of classes and wrappers determine each other. The appropriate domain of classes is simpler and will be discussed here.

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 of the current class is. applied to an existing class. Existing classes are either the empty class *Base* or a class resulting from the application of a wrapper to an existing class. The store remains unchanged because instance variables are not allocated at the time of class or wrapper declaration. An object is created by application of the fixed point operator to the class. For the fixed point operator to be applied to it, the domain of the class must be $\alpha \longrightarrow \alpha$ where α is any domain (the domain of classes was *Object* $\longrightarrow$ *Object* in section 2). The environment for methods is recursive whereas the environment for instance variables is not. We allocate the instance variables after applying 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 return it with the instance variables allocated; 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:

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

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

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

3.2.3. Semantic Clauses Continued

With the domain of classes, we are able to define object creation, one of the central clauses of this semantics:

---

[5] Note that for messages without arguments the empty parentheses can be omitted while for the method definitions they cannot.

[6] Including the instance variables declared in superclasses.

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

After evaluating E, we get a class. The fixed point operator Fix is applied to this class. The result of the application of Fix is applied to the current store s.[7] Fig. 8 contains object creations at the definition of p and c.

$$\mathsf{E}\ [\![\mathrm{E}_1\ \mathrm{O}\ \mathrm{E}_2]\!]\ \mathrm{r} \quad = \quad \mathsf{R}[\![\mathrm{E}_1]\!]\ \mathrm{r} \star \lambda \mathrm{e}_1.\mathsf{R}[\![\mathrm{E}_2]\!]\ \mathrm{r} \star \lambda \mathrm{e}_2.\mathsf{O}[\![\mathrm{O}]\!]\ (\mathrm{e}_1, \mathrm{e}_2)$$

The textually first binary operation in Fig. 8 is `X+self.x`.

$$\begin{array}{lcl}
\mathsf{C}[\![\mathrm{E}]\!]\ \mathrm{r} & = & \mathsf{E}[\![\mathrm{E}]\!]\ \mathrm{r} \\
\mathsf{C}[\![\mathrm{I} := \mathrm{E}]\!]\ \mathrm{r} & = & \mathsf{E}[\![\mathrm{I}]\!]\ \mathrm{r} \star \mathrm{Loc?} \star \lambda \mathrm{l}.\ \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} \star (\mathrm{update\ l}) \\
\mathsf{C}[\![\textbf{output } \mathrm{E}]\!]\ \mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} \star \\
 & & \lambda \mathrm{e}.\lambda \mathrm{s}.(\mathrm{u},\ [\mathrm{out} \mapsto \mathrm{append(s\ out,e)}] \oplus \mathrm{s}) \\
\mathsf{C}[\![\textbf{if } \mathrm{E} \textbf{ then } \mathrm{C}_1 \textbf{ else } \mathrm{C}_2]\!]\ \mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} \star \mathrm{Bool?} \star \\
 & & \mathrm{cond}(\mathsf{C}[\![\mathrm{C}_1]\!]\ \mathrm{r},\ \mathsf{C}[\![\mathrm{C}_2]\!]\ \mathrm{r}) \\
\mathsf{C}[\![\textbf{while } \mathrm{E} \textbf{ do } \mathrm{C}]\!]\ \mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r} \star \mathrm{Bool?} \star \\
 & & \mathrm{cond}(\mathsf{C}[\![\mathrm{C}]\!]\ \mathrm{r} \star \\
 & & \lambda \mathrm{e}.\mathsf{C}[\![\mathrm{while\ E\ do\ C}]\!]\ \mathrm{r},\quad \mathrm{result\ u}) \\
\mathsf{C}[\![\textbf{def } \mathrm{V} \textbf{ in } \mathrm{C}]\!]\ \mathrm{r} & = & \mathsf{V}[\![\mathrm{V}]\!]\ \mathrm{r} \star \lambda \mathrm{r}'.\mathsf{C}[\![\mathrm{C}]\!]\ (\mathrm{r}' \oplus \mathrm{r}) \\
\mathsf{C}[\![\mathrm{C}_1; \mathrm{C}_2]\!]\ \mathrm{r} & = & \mathsf{C}[\![\mathrm{C}_1]\!]\ \mathrm{r} \star \lambda \mathrm{e}.\mathsf{C}[\![\mathrm{C}_2]\!]\ \mathrm{r}
\end{array}$$

The result of an assignment-, an output-, or a while-expression is *unit*. In a sequence of expressions the transmitted values of all but the last expression are discarded. This practice has been adopted from ML [Mil84].

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

The result of evaluating a class declaration is the binding of a class to the class name. The store remains unchanged when a class is declared. The wrapper w takes an allocator for self-reference, an allocator 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, not in any subclass. Thus we have encapsulated instance variables.

[7] By applying the η-rule [Bar81, page 32] twice, $\lambda c.\lambda s.(\mathrm{Fix\ c})\mathrm{s}$ could be replaced by Fix.

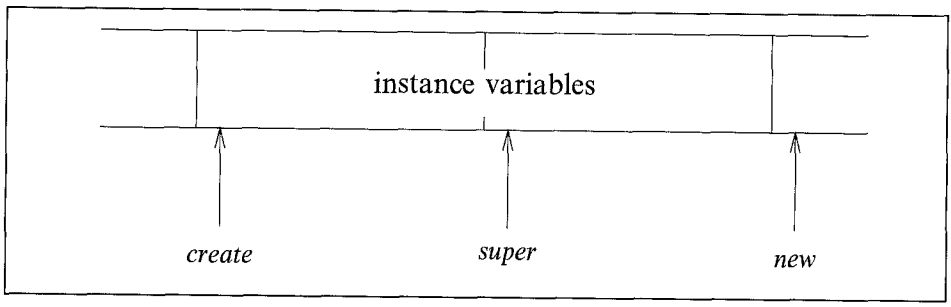

Fig. 9. The store during object creation.

Let us consider what happens at object creation time during the evaluation of the inner where-clause above. Fig. 9 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 of the allocator x_{self} 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.

Fig. 10 shows the effect on the store of the declaration `var c := new circle` (Fig. 8). We can see that the object c is allocated after its own instance variables.

$$K[\![K_1\ K_2]\!]\ r = K[\![K_1]\!]\ r\ \star\ \lambda r'.\ (K[\![K_2]\!]\ (r' \oplus r)) \oplus r'$$

$$K[\![\epsilon]\!]\ r = \text{result}\ [\,]$$

In Fig. 8, the second class definition knows the first but the first ignores the second. Nevertheless, it is possible, in principle, that an object of the first class sends messages to an object of the second class.

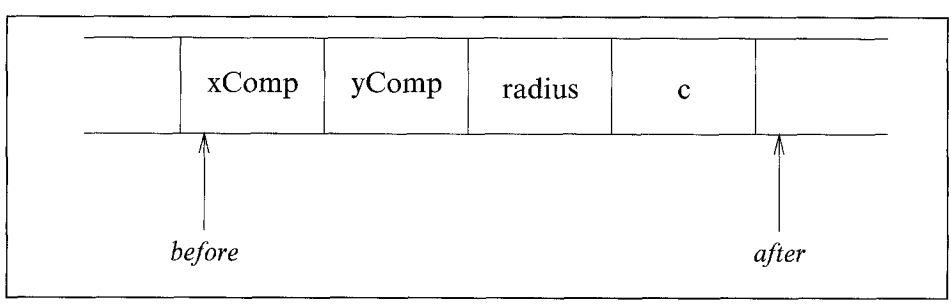

Fig. 10. The store after an object creation.

$$
\begin{array}{lcl}
\mathsf{V}[\![\textbf{var } \mathrm{I} := \mathrm{E}]\!]\ \mathrm{r} & = & \mathsf{R}[\![\mathrm{E}]\!]\ \mathrm{r}\ \star\ \lambda \mathrm{d}.\ \mathrm{new} \star \lambda \mathrm{l}.\lambda \mathrm{s}.\ ([\mathrm{I} \mapsto \mathrm{l}],\ [\mathrm{l} \mapsto \mathrm{d}] \oplus \mathrm{s}) \\
\mathsf{V}[\![\mathrm{V}_1\ \mathrm{V}_2]\!]\ \mathrm{r} & = & \mathsf{V}[\![\mathrm{V}_1]\!]\ \mathrm{r}\ \star\ \lambda \mathrm{r}'.\ (\mathsf{V}[\![\mathrm{V}_2]\!]\ (\mathrm{r}' \oplus \mathrm{r})) \oplus \mathrm{r}' \\
\mathsf{V}[\![\epsilon]\!]\ \mathrm{r} & = & \mathrm{result}\ [\,] \\
\mathsf{M}[\![\textbf{meth } \mathrm{I}(\mathrm{I}_1,\ldots,\mathrm{I}_n)\ \mathrm{C}]\!]\ \mathrm{r} & = & \left[\ \mathrm{I} \ \mapsto\ \ \lambda \mathrm{d}_1.\ \ldots\ \lambda \mathrm{d}_n.\ \mathrm{d}\ \right] \\
 & & \textbf{where} \\
 & & \mathrm{d} = \mathsf{C}[\![\mathrm{C}]\!]\left(\left[\begin{array}{lcl} \mathrm{I}_1 & \mapsto & \mathrm{d}_1 \\ & \vdots & \\ \mathrm{I}_n & \mapsto & \mathrm{d}_n \end{array} \right] \oplus \mathrm{r} \right) \\
\mathsf{M}[\![\mathrm{M}_1\ \mathrm{M}_2]\!]\ \mathrm{r} & = & (\mathsf{M}[\![\mathrm{M}_2]\!]\ \mathrm{r}) \oplus (\mathsf{M}[\![\mathrm{M}_1]\!]\ \mathrm{r}) \\
\mathsf{M}[\![\epsilon]\!]\ \mathrm{r} & = & [\,]
\end{array}
$$

The variable-declaration clause gets a new location in the store and puts the value of the right-hand side there. Method definitions are not recursive. Recursion and the calling of other methods is possible by sending messages to `self`.

3.3. Previous Work

The first semantics of Smalltalk [GoR89], which often serves as an archetype of object-oriented programming languages, was described operationally. The first denotational semantics of Smalltalk is due to Wolczko [Wol87]. His semantics still has some operational elements: inheritance is described by method lookup. A denotational Smalltalk-semantics in continuation style is from Kamin [Kam88]. Reddy [Red88] presents more readable semantics because he focuses on central issues of object-oriented programming; he uses fixed points for modeling *self*. Cook [Coo89, CoP89] gives a wrapper-semantics of inheritance without state.

4. Wrappers as a Language Construct

In object-oriented programming languages like Smalltalk or O'SMALL, a class declaration is a modification of an existing class. This existing class becomes the *superclass* of a new class. In some cases, the modification is interesting in its own right: it is advantageous to apply the modification to more than one superclass. Suppose we have already defined points and intend to add colour to them. In Smalltalk or classical O'SMALL, we would define a class of coloured points as a subclass of points. But now we also want coloured circles. Black-and-white circles are at our disposal. In Smalltalk or classical O'SMALL, we would define coloured circles as a subclass of circles. The colour-part (the colour-modification) must be defined twice. Therefore, we should be able to make the colour-modification *explicit* in the programming language. We call such modifications wrappers according to the semantic construct (sections 2 and 3).

Let us explain the difference between class declarations with and without explicit wrappers in the syntax of O'SMALL. In classical O'SMALL, a class *A* was defined as a subclass of another class *B*. The modification is contained in the

new instance variables V and the new methods M. In the following O'SMALL fragments, V represents a sequence of variable declarations and M represents a sequence of method declarations.

`class` A `subclassOf` B `def` V `in` M `ni`

Now, with explicit wrappers, a wrapper W is defined in much the same way as a class before, except that it does not name a superclass. The class definition defines A as the subclass of B that results from the modification by W.

`wrapper` W `def` V `in` M `ni` ..... `class` A = W B

The syntax and semantics of the modification, i.e. V and M, are as before. The class A is defined as the wrapper W applied to the superclass B. The two O'SMALL expressions separated by a dotted line define essentially the same as the previous class definition in classical O'SMALL. Now that wrappers are denotable, i.e. they can be bound to variables, they can be applied to several classes. For example:

`class` D = W C

Greater flexibility is thus gained.

4.1. Multiple Application of a Wrapper

4.1.1. Universal Wrappers

A wrapper having method names different from all method names in existing classes, whose method bodies refer to its own methods only using `self`, and with no occurrence of `super`, is called a *universal wrapper.*

Definition 4.1. For a wrapper definition W we define *outLabels(W)* as the set of method labels defined in W, *superLabels(W)* as the set of message-labels sent to the pseudo-variable `super` in W, and *selfLabels(W)* as the set of message-labels sent to the pseudo-variable `self` in W.

$$inLabels(W) = superLabels(W) \cup (selfLabels(W) - outLabels(W))$$

Now, we are able to define universality more precisely: a wrapper (definition) W is *universal* if and only if $inLabels(W) = \emptyset$ and $outLabels(W)$ is disjoint from all labels in other wrapper (definitions) in the program. A universal wrapper can be applied to any existing class. An example for a universal wrapper is the wrapper *COLOUR* in the O'SMALL program of Fig. 11.[8] We achieve an effect similar to hierarchy inheritance [Coo89], but we still have to apply the wrapper to each member of the hierarchy "by hand". The wrapper for the colour is applied to the class *Point* yielding the class *ColPoint*. The resulting class hierarchy can be seen in Fig. 12. *Point* is a superclass of *Circle*. The derived hierarchy is *ColPoint* and *ColCircle*. In the example, *ColCircle* is a subclass of *Circle*, but it could also be a subclass of *ColPoint*, as the dashed line suggests. That is to say, had we applied *CIRCLE* to *ColPoint* this would have resulted in the same *ColCircle*.[9] We state the following without proof:

[8] In our O'SMALL examples wrapper identifiers are written in upper case and class identifiers begin with uppercase letters.

[9] In a "real language" it should be possible to identify any universal wrapper with the class that results from its application to the empty class. This would economize on writing.

```
wrapper POINT
def var xComp := 0; var yComp := 0
in meth x()                 xComp
   meth y()                 yComp
   meth move(X,Y)           xComp := X+self.x; yComp := Y+self.y
   meth distFromOrg()       sqrt(sqr(self.x) + sqr(self.y))
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg
ni

wrapper CIRCLE
def var radius := 0
in meth r()            radius
   meth setR(d)        radius := d
   meth distFromOrg() max(0, super.distFromOrg - self.r)
ni

wrapper COLOUR
def var c := 1
in meth setColour(t) c := t
   meth colour()     c
ni

class Point     = POINT Base          { Base is the empty class }
class Circle    = CIRCLE Point
class ColPoint  = COLOUR Point
class ColCircle = COLOUR Circle
```

Fig. 11. Universal wrapper.

Proposition 4.2. Commutativity of wrapper application:
If

$$\begin{aligned}\emptyset &= outLabels(W_1) \cap outLabels(W_2)\\ &= inLabels(W_1) \cap outLabels(W_2)\\ &= inLabels(W_2) \cap outLabels(W_1)\end{aligned}$$

then $[\![W_1]\!] \boxed{\triangleright} [\![W_2]\!] \boxed{\triangleright} Base = [\![W_2]\!] \boxed{\triangleright} [\![W_1]\!] \boxed{\triangleright} Base.$

Corollary 4.3. The application of universal wrappers is commutative.

4.1.2. Special Wrappers

A wrapper is called *special* if it is not universal. A special wrapper refers, somehow, to methods defined in its superclass. If it were applied to a class that does not define the methods in the expected way, errors might result. We will show an example of a special wrapper that can, nevertheless, be applied to more than one class.

Definition 4.4. Let M be a set. A subset $H \subset M \times M$ defines a *preorder* (we often write $a \leq b$ for $(a,b) \in H$), if $\forall a,b,c \in M$:

$$a \leq a, \qquad a \leq b \ \wedge \ b \leq c \Rightarrow a \leq c$$

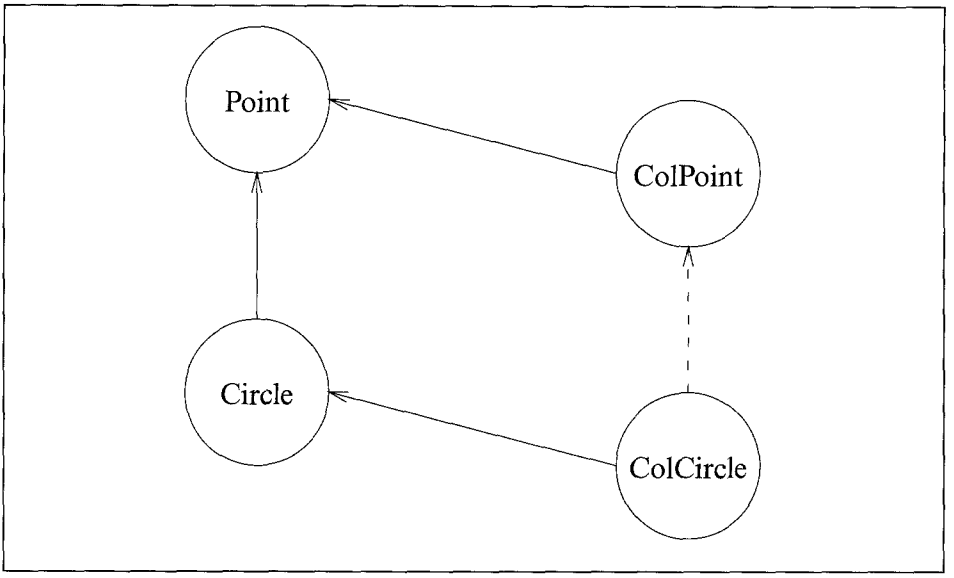

Fig. 12. Hierarchy inheritance.

If, in addition, $a \leq b \wedge b \leq a \Rightarrow a = b$ holds, $(M, . \leq .)$ is called a *(partial) order.*

Let $\leq$ be a relation defined on $\mathbf{N} \times \mathbf{N}$ as: $(x, y) \leq (x', y') \Leftrightarrow x + y \leq x' + y'$.
Let $\leq$ be a relation defined on $\mathbf{Z}$ as: $z \leq z' \Leftrightarrow \exists x \in \mathbf{Z} : z * x = z'$
One easily verifies that $(\mathbf{N} \times \mathbf{N}, . \leq .)$ and $(\mathbf{Z}, . \leq .)$ are preorders. It is possible to obtain an order from every preorder by using equivalence classes, where two elements a and b of a preorder are equivalent $(a \approx b)$ if $a \leq b \wedge b \leq a$. An order is obtained by regarding $\approx$ as the equality.

The O'SMALL program of Fig. 13 shows the two previous preorders and a wrapper that uses equivalence classes and, thus, makes an order from every preorder. In the definition of the wrapper `PREORDER2ORDER`, we have chosen `eq` for the equality and `leq` for the relation. Obviously, the wrapper `PREORDER2ORDER` only makes sense with classes where this naming convention (signature) is respected.

4.2. Multiple Inheritance

One speaks of *multiple inheritance* when a class directly inherits the properties of at least two classes. This implies, according to our view, that objects of the new class may be substituted for objects of both parent classes. There is a problem when there are name conflicts [Knu88] between the inherited classes.

Let A and B in Fig. 14 each define a method m, and let m not be redefined in C. Let us denote by m_A the definition in A and by m_B the definition in B. If m_A and m_B are incompatible at the signature level, m_A may, for example, require two parameters and m_B three, cancellation at the signature level is the consequence because either m_A or m_B has to be chosen when a message with the selector m is sent to an object of class C. One can try to overcome the name conflict by renaming [Mey88] but this is no remedy for cancellation. Cancellation at the signature level is not desirable because it may be a cause of errors if one assumes

```
wrapper PAIR
def var xComp := 0    var yComp := 0
in meth set(a,b) xComp := a; yComp := b
   meth x()      xComp
   meth y()      yComp
   meth leq(p)   (self.x+self.y) <= (p.x+p.y)
   meth eq(p)    self.x=p.x and self.y=p.y
ni

wrapper DIV
def var z := 1  {should not be 0 because of 'mod'}
in meth set(v)  z:=v
   meth value() z
   meth leq(n)  (n.value mod |self.value|) = 0
   meth eq(n)   self.value = n.value
ni

wrapper PREORDER2ORDER
   meth eq(e) self.leq(e) and e.leq(self)

class Pair        = PAIR Base    {Base is the empty class}
class Div         = DIV Base
class OrderedPair = PREORDER2ORDER Pair
class OrderedDiv  = PREORDER2ORDER Div
```

Fig. 13. A special wrapper.

that objects of a superclass may be substituted by objects of a subclass. If m_A and m_B are compatible at the behavioural level (this implies compatibility at the signature level) there is no cancellation. Still, either m_A or m_B has to be chosen.

In every case described so far, the inheritance graph in Fig. 14 is either impossible, because the resulting C cannot be signature-compatible with both A and B, or it is misleading, because the graph suggests symmetry where there is none. Fig. 14 is acceptable only if every method for a message selector that is understood by A and B is defined in a common superclass of A and B. That is to say, we regard Fig. 14 just as a graphical way to express that C inherits the properties of A and B (with single inheritance), but it does not matter in which order. A sufficient condition for commutativity has been given in proposition 4.2.

All cases of disciplined multiple inheritance, where the symmetry of Fig. 14 is justified, can now be defined with single inheritance and explicit wrappers. For every class, there is a defining sequence of wrappers starting from the predefined class *Base* (see section 3). Therefore, we can represent A and B as

$$A = W_l^A \boxed{\triangleright} \cdots W_1^A \boxed{\triangleright} Base,$$
$$B = W_m^B \boxed{\triangleright} \cdots W_1^B \boxed{\triangleright} Base.$$

Wrappers are now explicitly denotable values, so we can reuse them and define C as

$$C = W^C \boxed{\triangleright} W_l^A \boxed{\triangleright} \cdots W_1^A \boxed{\triangleright} W_m^B \boxed{\triangleright} \cdots W_1^B \boxed{\triangleright} Base.$$

Fig. 15 shows a special case of Fig. 14 with iceberg diagrams. A is derived from

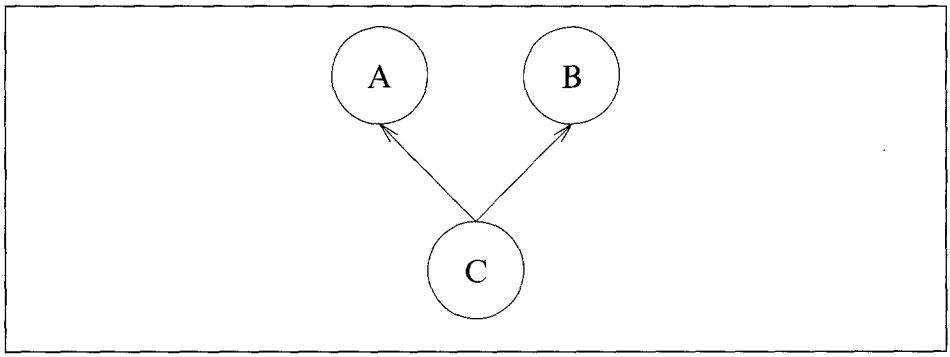

Fig. 14. Multiple inheritance.

Base with two wrappers ($l = 2$), B with three ($m = 3$), and C by multiple inheritance without further definitions.

If A and B in Fig. 14 have a common superclass other than *Base* their wrappers are not disjoint: there exists a k such that, for all $1 \leq i \leq k$, $W_i^A = W_i^B$. There may be problems with the internal state: let us assume that, in these common wrappers, we have defined a counter. The resulting class C contains two counters, one open and one hidden. We call the counter defined by the W_i^A "open" because it is the one that is accessed in the above definition of C. The counter defined by the W_i^B is called "hidden" because it is "overwritten" by the W_i^A. In most cases, the open counter will be used all the time and its associated instance variables will contain its state. There is, however, one way to use the hidden counter: in a method of B that is not defined in A a counter method, say f, is called with `super.f`. Usually, `super.f` is only used if we have overwritten f and still want to refer to the old definition. As pointed out above, this should not be the case for multiple inheritance: we should *not* have overwritten f in A or B because of cancellation. This example shows that multiple inheritance modeled with single inheritance and explicit wrappers is not a simple automatic procedure. In the general case, one has to be careful and take consequences like this into consideration.

The denotational semantics of section 3 can easily be extended by explicit wrappers. The details are explained in section B.

4.3. Related Work

Snyder [Sny86] categorizes different strategies in multiple inheritance into *graph oriented solutions* as in TRELLIS/OWL [SCB86], *linear solutions* as in FLAVORS [Moo86] or COMMONLOOPS [BKK86], and *tree solutions* as in Common Objects [Sny85]. In graph-oriented solutions, the inheritance graph is searched directly (in terms of method-lookup-semantics) by a depth-first-traversal for example. Instance variables of ancestors that can be reached by more than one inheritance path are not duplicated. Graph-oriented solutions are flexible yet complicated, and they make inheritance become part of the external interface [Sny86]. In linear solutions, the inheritance graph is transformed into a chain. The problem here is that one class may have a new parent which the designer was not aware of. In tree solutions the graph is transformed into a tree by duplicating nodes.

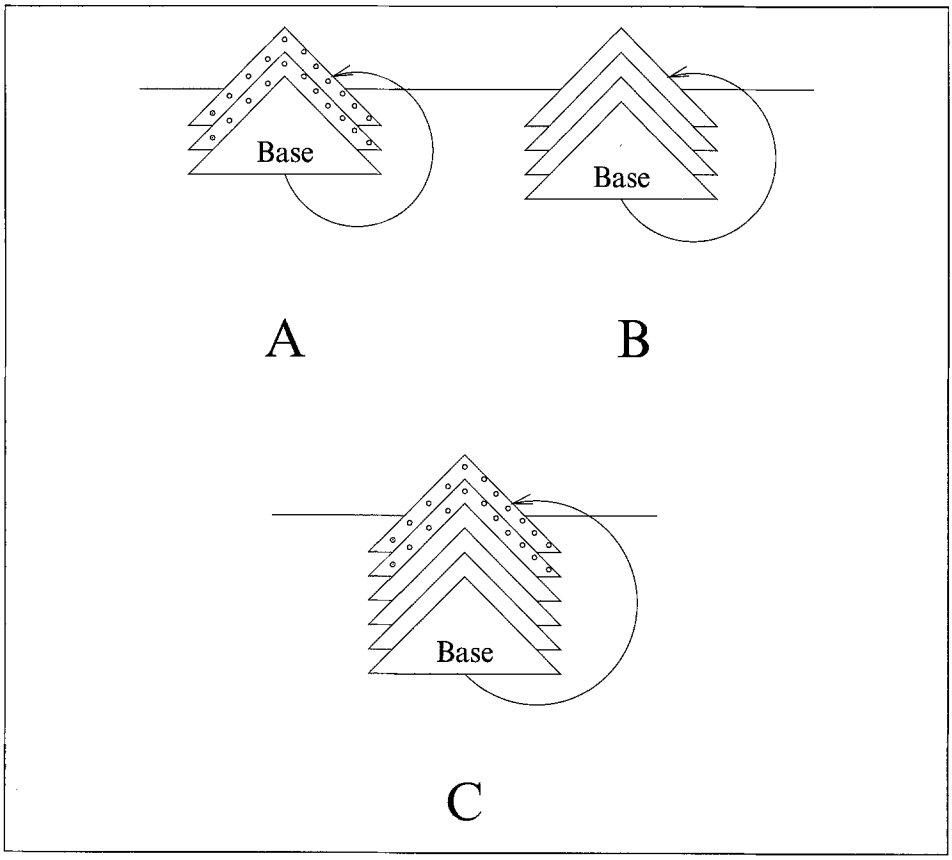

Fig. 15. Multiple inheritance.

This implies duplication of the instance variables. For each inheritance path to a superclass, a new set of instance variables for that superclass is created. Our solution may be seen as a linear solution. In O'SMALL there are no name-conflicts with instance variables because they are encapsulated.

The mixing of *flavors* in the programming language FLAVORS [Moo86] resembles our explanation of multiple inheritance most. Flavors can be regarded as analogous to classes that can be used like wrappers. This comes from another context and there are some differences. FLAVORS has intricate method combination, and instance variables are not encapsulated. Duplicate flavors in multiple inheritance are eliminated. This would correspond to the elimination of duplicates of identical wrappers $W_i^A = W_j^B$ in the definition of C. In our context this is impossible in general because of the pseudo-variable `super`. Whereas FLAVORS uses a standard mechanism for multiple inheritance, O'SMALL enables the simulation of multiple inheritance "by hand". In O'SMALL, it is the programmer's responsibility to solve name conflicts.

Cook [Coo89] treats multiple inheritance with so called n-wrappers. n-wrappers generalize wrappers by taking a tuple of superclasses instead of just one. The one elaborated construction with n-wrappers is *strict multiple record inheritance*: parents are combined so that conflicting methods are removed. Con-

flicting methods should be redefined. The new class definition has direct access to all parent-class-methods. Other constructions are possible. n-wrappers are more general than our approach.

There is a relationship between explicit wrappers and abstract parameterized data types and modules [EhM85, EhM90]. The issue of inheritance from the point of view of algebraic specification is addressed by [ESS89]. Modules, on a programming language level, are most advanced in ML [Mac85]. In ML modules are called structures. Functors in ML generate new structures by modifying existing structures. Classes correspond to ML-structures and wrappers correspond to ML-functors. In contrast to wrappers, functors can have more than one argument and arguments must be qualified; this is similar to n-wrappers. The main difference is the absence of late binding in ML. If we depict functors and structures in the same way as wrappers and classes, with iceberg diagrams, the difference between ML and O'SMALL can be seen in Fig. 16. ML has no pseudo-variable `self`, but it has recursion. Thus, the arrows in the diagram stand for recursion in the functions of the structure. A function of the old (lower) structure will always recursively refer to functions of the old structure, even if functions of the same name have been defined in the functor (*static binding*).

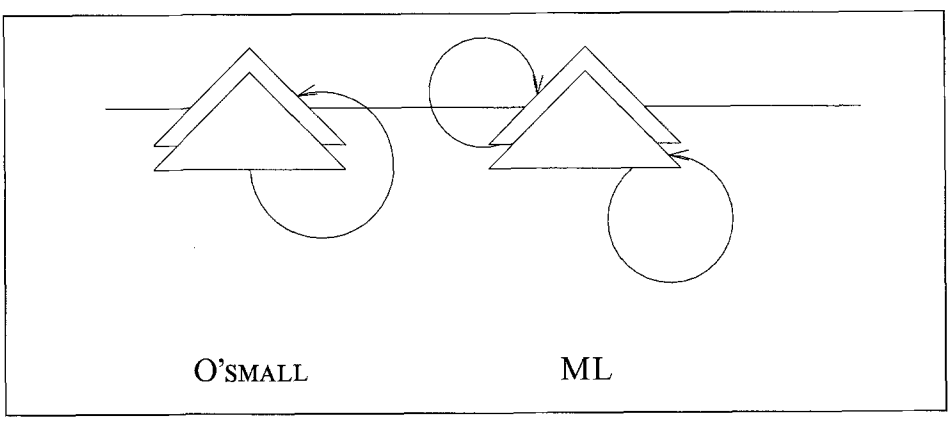

Fig. 16. Late binding/static binding.

Explicit wrappers have been invented independently by Bracha and Cook [BrC90]. There, they are called *mixins*. Their language is an extension of Modula-3 [CDG89]. Modula-3 has been chosen because it supports single inheritance and is strongly typed. We agree that explicit wrappers necessitate strong typing even more than ordinary inheritance. We have investigated static type inference for O'SMALL [Hen93]. Further examples that show the usefulness of explicit wrappers are contained in [BrC90].

Wrappers have nothing in common with continuations. However, the process of lifting semantic constructs to the programming language level has been performed in SCHEME [ASS85] and recently in ML with explicit continuations.

5. Conclusion

We have proposed explicit wrappers as a new language feature for object oriented-programming. To make this article self-contained, we have included the semantics of classical O'SMALL. This semantics shows one way of extending wrapper-semantics [Coo89] by state. Thus, the vehicle of our examples has a concise formal definition, including the extension of O'SMALL by explicit wrappers. We have shown three uses of explicit wrappers, namely

- hierarchy inheritance, where the wrappers have to be applied one by one to classes of the original hierarchy,
- functor-like application for transformation of mathematical structures, and
- (a variety of) multiple inheritance.

Notions like disciplined inheritance and properties of explicit wrappers have remained informal. A theory of explicit wrappers is a prerequisite for their future use. With single inheritance and explicit wrappers, we are not able to model all existing varieties of multiple inheritance but we are able to model disciplined multiple inheritance. If disciplined multiple inheritance is a strong enough mechanism in practice, is an open question. A disciplined programming style, assisted by ameliorated static analysis, seems to be appropriate and at the same time a direction of future research.

Acknowledgements

Thanks to Christian Fecht, Andreas Gündel, Reinhold Heckmann, Joachim Philippi, Reinhard Wilhelm, and Mario Wolczko for instructive discussions and valuable comments on earlier drafts.

References

[ASS85] Abelson, H., Sussman, G.J. and Sussman J.: *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Bar81] Barendregt H.P.: *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics.* North-Holland, revised 1984 edition, 1981.

[BrC90] Bracha G. and Cook W.: Mixin-based inheritance. In *Object-Oriented Programming Systems, Languages and Applications*, pages 303–311. ACM, October 1990. European Conference on Object-Oriented Programming.

[BKK86] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F. and Xerox Parc.: CommonLoops: Merging lisp and object-oriented programming. In *Object-Oriented Programming Systems, Languages and Applications*, pages 17–29. ACM, September 1986.

[CDG89] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. and Nelson, G.: Modula-3 report (revised). Technical Report 52, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, November 1989.

[CHC90] Cook, W., Hill, W. and Canning, P.: Inheritance is not subtyping. In *Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, January 1990. ACM.

[Coo89] Cook W.R.: A denotational semantics of inheritance. Technical Report CS-89-33, Brown University, Dept. of Computer Science, Providence, Rhode Island 02912, May 1989.

[CoP89] Cook, W. and Palsberg, J.: A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming Systems, Languages and Applications*, pages 433–444. ACM, October 1989.

[EhM85] Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification*, volume 1: Equations and Initial Semantics of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[EhM90] Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification*, volume 2: Module Specifications and Constraints of *EATCS Monographs on Theoretical Computer Science*. Springer, 1990.

[ESS89] Ehrich, H.D., Sernadas, A. and Sernadas, C.: From data types to object types. Braunschweig, 1989.

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

[GoR89] Goldberg, A. and Robson, D.: *Smalltalk-80: the Language*. Addison-Wesley, 1989.

[Gün90] Gündel, A.: Compatibility conditions on subclasses. Unpublished notes, Universität Dortmund, 1990.

[Hen90] Hense, A.V.: Denotational semantics of an object-oriented programming language with explicit wrappers. Technical Report A 11/90, Universität des Saarlandes, Fachbereich 14, June 1990.

[Hen91] Hense, A.V.: Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, September 1991.

[Hen93] Hense, A.V.: *Polymorphic type inference for object-oriented programming languages*. PhD thesis, Universität des Saarlandes, Fachbereich 14, D-6600 Saarbrücken, 1993. forthcoming.

[HiS86] Hindley, J.R. and Seldin, J.P.: *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

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

[Knu88] Knudsen, J.L.: Name collision in multiple classification hierarchies. *Lecture Notes in Computer Science*, 322:93–109, 1988. European Conference on Object-Oriented Programming.

[Mac85] MacQueen, D.: Modules for standard ML. In *Polymorphism The ML/LCF/Hope Newsletter*, 1985. Vol. II, No.2.

[Mey88] Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall, 1988.

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

[Moo86] Moon, D.A.: Object-oriented programming with Flavors. In *Object-Oriented Programming Systems, Languages and Applications*, pages 1–8. ACM, September 1986.

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

[SCB86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M. and Wilpolt, C.: An introduction to Trellis/Owl. In *Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, 1986. ACM.

[Sco76] Scott, D.S.: Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.

[Sny85] Snyder, A.: Object-oriented programming for Common Lisp. Technical Report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985.

[Sny86] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In *Object-Oriented Programming Systems, Languages and Applications*, pages 38–45. ACM, September 1986.

[Tar55] Tarski, A.: A lattice-theoretical fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Wol87] Wolczko, M.: Semantics of Smalltalk-80. *Lecture Notes in Computer Science*, 276:108–120, 1987. European Conference on Object-Oriented Programming.

[WeZ88] Wegner, P. and Zdonik, S.: Inheritance as an incremental modification mechanism or what like is and isn't like. *Lecture Notes in Computer Science*, 322:55–77, August 1988. European Conference on Object-Oriented Programming.

Appendices

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 types

$$f \;:\; \left\langle \begin{array}{c} Store \\ D_1 \longrightarrow Store \end{array} \right\rangle \longrightarrow [D_2 \times Store],$$

$$g \;:\; D_2 \longrightarrow Store \longrightarrow [D_3 \times Store].$$

The lines inside the angular delimiters represent alternatives. The alternatives in the following text are not free but depend on the choices of the alternatives above: if in the above delimiters one chooses the upper alternative, one must choose the upper alternative below (and vice versa). The composition of f and g has type

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

and is defined by

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

$$\text{where} \qquad (d_2, s_2) = \left\langle \begin{array}{c} f s_1 \\ f d_1 s_1 \end{array} \right\rangle$$

The infix operator $\star$ is left associative. The definition of $\triangleright$ in section 2 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,

$$\begin{aligned} x_1 \oplus x_2 \;=\; & \lambda s.(r_1 \oplus_{lpr} r_2, s') \\ & \text{where} \\ & (r_1, s') = x_1 s, \\ & (r_2, \_) = x_2 s. \end{aligned}$$

The infix operator $\oplus_{lpr}$ stands for the operation on records that is defined in definition 2.2. This is the only change of the inheritance function (definition 1). 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

$$\text{D?} = \lambda \text{d.} \begin{cases} \text{result d, if isD d} \\ \text{seterr , otherwise} \end{cases}$$

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

$$\text{deref} = \lambda d.\begin{cases} \text{cont d , if isLoc d} \\ \text{result d, otherwise} \end{cases}$$

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? ⋆ λd.λs.(unit, [l ↦ d] ⊕ s)

B. From Classical O'SMALL to O'SMALL with Explicit Wrappers

The replacement purely technical. Replace the compound syntactical domain *Cla* by

WrCl the domain of wrapper and class declarations W

In the syntactic clauses replace P and K by

P ::= W C
W ::= **wrapper** I **def** V **in** M | **class** $I_1 = I_2\ I_3$ | $W_1\ W_2$ | ϵ

Add wrapper values to the denotable values in the domain equations:

Dv = Loc + Rv + Method_n + Class + Wrapper denotable values d

Replace the semantic function K at all its occurrences by

W : WrCl⟶ Env⟶ Store⟶ [Env×Store]

Replace the clauses for class definitions by the following two clauses:

W ⟦**wrapper** I **def** V **in** M⟧ r = result [I ↦ w]
where w = $\lambda x_{self}.\lambda x_{super}.\lambda s_{create}.$

$$(M[\![M]\!]\ (\begin{bmatrix} \text{self} & \mapsto & r_{self} \\ \text{super} & \mapsto & r_{super} \end{bmatrix} \oplus r_{local} \oplus r)\ ,\ s_{new})$$

$(r_{super}, s_{super}) = x_{super}\ s_{create}$
$(r_{local}, s_{new}) = V[\![V]\!]\ r\ s_{super}$
$(r_{self}, \_) = x_{self}\ s_{create}$

and

W ⟦**class** $I_1 = I_2\ I_3$⟧ r =
E⟦I_3⟧ r ⋆ Class? ⋆ λc. E⟦I_2⟧ r ⋆ Wrapper? ⋆ λw. result[I_1 ↦ w ▷ c]

I_2 denotes a wrapper and I_3 denotes a class. The identifiers are looked up in the environment, and the result of the wrapper application is bound to I_1. The result of the evaluation of a class declaration is the binding of a class to the class name.

$$\mathsf{W}[\![\mathsf{W}_1\ \mathsf{W}_2]\!]\ \mathsf{r} = \mathsf{W}[\![\mathsf{W}_1]\!]\ \mathsf{r}\ \star\ \lambda \mathsf{r}'.\ (\mathsf{W}[\![\mathsf{W}_2]\!]\ (\mathsf{r}' \oplus \mathsf{r})) \oplus \mathsf{r}'$$
$$\mathsf{W}[\![\epsilon]\!]\ \mathsf{r} = \mathsf{result}\ []$$

The class definition of classical O'SMALL has been removed and can now be reintroduced as syntactic sugar.

Received May 1991
Accepted in revised form September 1992 by T.S.E. Maibaum