Denotational Semantics of an Object Oriented Programming Language with Explicit Wrappers

Andreas V. Hense

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 11/90

revised version

Denotational Semantics of an Object Oriented Programming Language with Explicit Wrappers

Andreas V. Hense

Lehrstuhl für Programmiersprachen
und Übersetzerbau
FB–14 Informatik
Universität des Saarlandes
6600 Saarbrücken 11
Fed. Rep. of Germany
e-mail: hense@cs.uni-sb.de

February 15, 1991

Abstract

Wrappers are a mechanism in denotational semantics that model class inheritance of object oriented programming. In this paper we try to give evidence that the unusual step of reintroducing a semantic mechanism into the language being described can be sensible.

With wrappers now being explicit, a disciplined variant of multiple inheritance can be formulated as single inheritance and a better reusability of code is gained.

keywords: wrapper, denotational semantics, class inheritance, multiple inheritance, hierarchy inheritance

Besides the need for denotational semantics for conducting proofs and defining a programming language abstractly, a denotational semantics may help the novice in understanding the principles of a new programming language if the semantics is written in a clear and intuitive way. Advocating denotational semantics for teaching stems from the experience that "operational reasoning is a tremendous waste of mental effort" [Dij89, page 1403]. Of course, this can be argued and many people will think that operational semantics is more intuitive. The naturalness of semantics is a matter of personal preferences, after all.

Smalltalk [GR89] often serves as a prototype of object oriented programming languages. The first semantics for Smalltalk 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 has been overcome by [Red88] who described a small object oriented programming language with a direct semantics; he uses fixed points for modeling *self*. [Wol88] regards inheritance as an optional feature. He gives a denotational semantics of a language with objects and classes. The language with inheritance is then translated into a language without inheritance. [CP89] described the semantics of inheritance without state using wrappers. According to [Weg87] an essential feature of object oriented programming are objects *with state*. [Hen90] showed that wrapper semantics is applicable to an object oriented programming language with state.

Once a denotational semantics using wrappers is defined for a language, it is easy to introduce explicit wrappers into the language: informally, on the programming language level you just have to separate the definition of a new class from the naming of a superclass. This step is exemplified in O'small. The separation results in an increased flexibility: in addition to reusing classes one can reuse the differences that discern classes from each other.

Section 1 defines wrappers in a purely functional framework (objects have no state). Section 2 discusses explicit wrappers and gives motivating examples in O'small. Among them are multiple inheritance and the application of one wrapper to different classes. The appendix A contains the denotational semantics of O'small.

1 Semantics of Inheritance

This section describes the semantics of inheritance *without state* as in [Coo89].

Definition 1.1 A *record* is a finite mapping from a set of labels onto a set of values. A record is denoted by $\begin{bmatrix} x_1 & \mapsto & v_1 \\ & \vdots & \\ x_n & \mapsto & v_n \end{bmatrix}$ 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 [].

An *object* is a record with functions as values. These functions my refer recursively to the whole record. Note that in this section objects have no state. 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:

$$FACT \;=\; \lambda s.\lambda n.\ \text{if } n = 1 \text{ then } 1 \text{ else}$$

Let $\mathbf{Y}$ denote the fixed point operator. 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. C is the class of points in the plane. The variable s is used like *self.*

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

An object p of class C is created by:

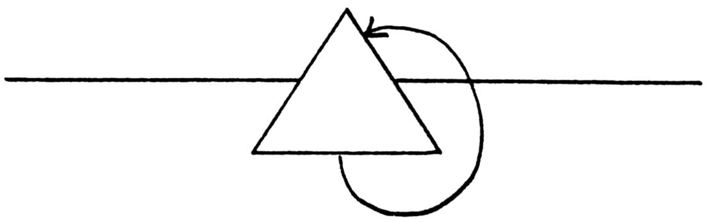

Figure 1: class

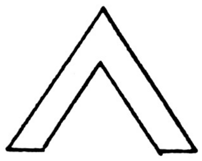

Figure 2: wrapper

$$p = \mathbf{Y}(C\ 2\ 2) = \begin{bmatrix} x & \mapsto & 2 \\ y & \mapsto & 2 \\ \mathit{distFromOrg} & \mapsto & \sqrt{8} \\ \mathit{closerToOrg} & \mapsto & \lambda p.\sqrt{8} < p.\mathit{distFromOrg} \end{bmatrix}$$

Inheritance is the derivation of a new generator from an existing one, where 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.

Definition 1.2 Let $dom(m) = \{x \mid m(x) \neq \perp\}$. The *left-preferential combination of records* is defined by:

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

Definition 1.3 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 1.4 The *inheritance function* $\boxed{\triangleright}$ applies a wrapper W to a class C and returns a class. $\triangleright$ is defined by: $w \triangleright c = (w \cdot c) \oplus c = w(c) \oplus c$ where $\cdot$ is the application.

The wrapper W specifies the differences between points an circles. The variable p is used like *super*.

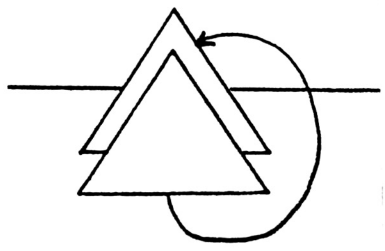

Figure 3: inheritance

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

The circle-class C' is created by:

$$C' \;=\; \lambda a.\lambda b.\lambda c.(W\ a\ b\ c) \boxed{\triangleright} (C\ a\ b)$$

An object c of class C' is created by:

$$c = \mathbf{Y}(C'\ 3\ 3\ 2) = \begin{bmatrix} x & \mapsto & 3 \\ y & \mapsto & 3 \\ r & \mapsto & 2 \\ distFromOrg & \mapsto & \sqrt{18} - 2 \\ closerToOrg & \mapsto & \lambda p.\sqrt{18} - 2 < p.distFromOrg \end{bmatrix}$$

To illustrate the inheritance function we introduce iceberg-diagrams, an intuitive description of classes and wrappers. A class is depicted by a triangle where the visible part (methods in O'small) is above the surface and the invisible part (instance variables in O'small) below (figure 1). There is an arrow for self-reference. A wrapper is depicted by an angular shape (figure 2). Inheritance is a wrapper applied to a class (see figure 3). 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.

2 Wrappers as a Language Construct

In object oriented programming languages with class inheritance, e.g. Smalltalk [GR89], a class declaration is a modification of an existing class, its superclass. In some cases the modification is interesting in its own right: it is advantageous to apply the modification to more than one superclass. We call such modifications wrappers according to the semantic construct of [Coo89].

Let us explain the difference between class declarations with and without explicit wrappers in the syntax of the object oriented programming language O'small. Before, i.e. in [Hen90], 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.

```
wrapper W def V in M ni
```

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 next program line is again O'small-syntax.[1]

```
class A = W B
```

But 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
```

2.1 Multiple Application of a Wrapper

2.1.1 Universal Wrappers

A wrapper with method names different from all method names in existing classes, whose method bodies refer with *self* to its own methods only, and with no occurrence of *super*, is called a *universal wrapper.* A universal wrapper can be applied to any existing class. An example for a universal wrapper is the wrapper *COLOR* in the O'small-program in figure 4.[2] This is what [Coo89] calls hierarchy inheritance – although you still have to apply the wrapper to each member of the hierarchy "by hand". The wrapper for the color is applied to the class *Point* which results in the class *ColPoint*. The resulting class hierarchy can be seen in figure 5. *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*.[3]

2.1.2 Special Wrappers

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

[1] If we were allowed to go beyond ASCII in a programming language we might have replaced the program line by *class* $A = W \boxed{\triangleright} B$.

[2] in our O'small-examples wrapper identifiers are written in upper case and class identifiers starting with capital letters.

[3] 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+xComp; yComp := Y+yComp
   meth distFromOrg()       sqrt(xComp*xComp + yComp*yComp)
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg ni

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

wrapper COLOR
def var c := 1
in meth setColor(t) c := t
   meth color()     c ni

class Point     = POINT Base          class Circle    = CIRCLE Point
class ColPoint  = COLOR Point         class ColCircle = COLOR Circle
```

Figure 4: universal wrapper

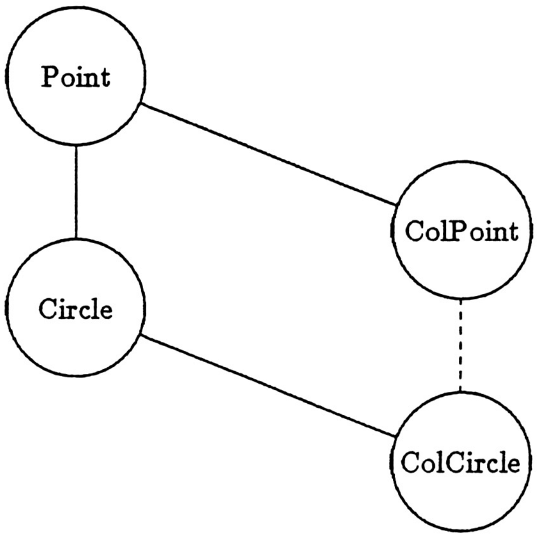

Figure 5: hierarchy inheritance

Definition 2.1 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 the following holds:

$$\forall a \in M : \; a \leq a$$
$$a \leq b \;\wedge\; b \leq c \;\Rightarrow\; a \leq c$$

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

Definition 2.2 Let $\mathbb{N} \times \mathbb{N}$ be the set and $\leq$ a relation defined as:

$$(x,y) \leq (x',y') \;\Leftrightarrow\; x + y \;\leq\; x' + y'$$

Definition 2.3 Let $\mathbb{Z}$ be the set and $\leq$ a relation defined as:

$$z \leq z' \;\Leftrightarrow\; \exists x \in \mathbb{Z} : z * x = z'$$

One easily verifies that $(\mathbb{N} \times \mathbb{N}, . \leq .)$ and $(\mathbb{Z}, . \leq .)$ are preorders. It is possible to obtain an order from every preorder by changing to 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 in figure 6 shows the preorders of definition 2.3 and definition 2.2 and a wrapper that changes to the equivalence classes and thus makes an order from every preorder. We require the equality to be named *eq* and the relation *leq*.

2.2 Multiple Inheritance

Inheritance is a mechanism for incremental modification. In our current framework it is possible to redefine methods such 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 run time errors will occur. But we require more than that and advocate a disciplined version of inheritance that allows us to make 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. Therefore inheritance should exclude cancellation. Cancellation does not occur at the level of names in O'small, i.e. every method of the superclass must occur in the subclass. For inheritance, as explained in section 1, to work, a minimal type compatibility is required: the type of the self-reference of an inheritor must be a subtype [Car88] of the type of the self-reference of its parents [CHC90]. For the principle of substitutability [WZ88] modulo a static type checking system also type compatibility of the external interfaces (signature compatibility) of inheritor and parents is necessary. Flexible static type checking systems for object oriented programming languages are subject to research. Semantic compatibility (behavioral compatibility) is also desirable. There has been a pragmatic approach with Hoare-logic where some aspects of semantic compatibility are checked at run-time [Mey88]. The definition of semantic compatibility in object oriented languages is still subject to research [Gun90].

One speaks of multiple inheritance when a class inherits the properties of at least two classes. This implies, according to our view, that objects of the new class are substitutable for objects

```
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)   (xComp+yComp) <= (p.x+p.y)
   meth eq(p)    xComp=p.x and yComp=p.y ni

wrapper DIV
def var z := 0
in meth set(v)  z:=v
   meth value() z
   meth leq(n)  (n.value mod z) = 0
   meth eq(n)   z = n.value ni

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

class Pair        = PAIR Base
class Div         = DIV Base
class OrderedPair = PREORDER2ORDER Pair
class OrderedDiv  = PREORDER2ORDER Div
```

Figure 6: a special wrapper

of both parent classes. There is a problem when there are name conflicts [Knu88] between the classes from which is inherited.

Let A and B in figure 7 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, if they have incompatible types, cancellation at the signature level is the consequence because either m_A of 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 master 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 run time errors. If m_A and m_B are compatible at the behavioral 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 figure 7 is either impossible, because the resulting C is illegal, or misleading, because the graph suggests symmetry where there is none. Figure 7 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 figure 7 only 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.

After tailoring multiple inheritance to our needs, we are able to define it with explicit wrappers. For every class there is a defining sequence of wrappers, starting from the predefined class

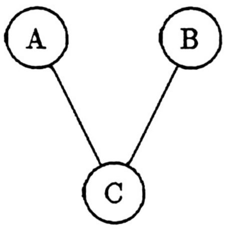

Figure 7: multiple inheritance

Base (see section A). We can therefore represent A and B like:

$$\begin{aligned} A &= W_l^A \boxed{\triangleright} (\cdots(W_1^A \boxed{\triangleright} \mathit{Base})\cdots) \\ B &= W_m^B \boxed{\triangleright} (\cdots(W_1^B \boxed{\triangleright} \mathit{Base})\cdots) \end{aligned}$$

Because wrappers are now explicitly denotable values we can reuse them and define C as:

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

Let us look at an example represented by iceberg diagrams. Let in figure 7, A be derived from *Base* with two wrappers ($l = 2$) and B with three ($m = 3$). Let further the two sets of wrappers be disjoint. If C does not add any definitions, the multiple inheritance for C is depicted by figure 8.

2.3 Related work

[Sny86] categorizes different strategies in multiple inheritance into *graph oriented solutions* as in Trellis/Owl [SCB*86], *linear solutions* as in Flavors [Moo86] or CommonLoops [BKK*86], and *tree solutions* as in Common Objects [Sny85]. 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 of which the designer was not aware of. In tree solutions the graph is transformed into a tree by duplicating nodes. 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 responsibility of the user to solve name conflicts.

[Coo89] characterizes multiple inheritance in a general way and not completely. He introduces a new kind of wrapper (n-wrapper) whereas we use ordinary wrappers. n-wrappers are more general than our wrappers.

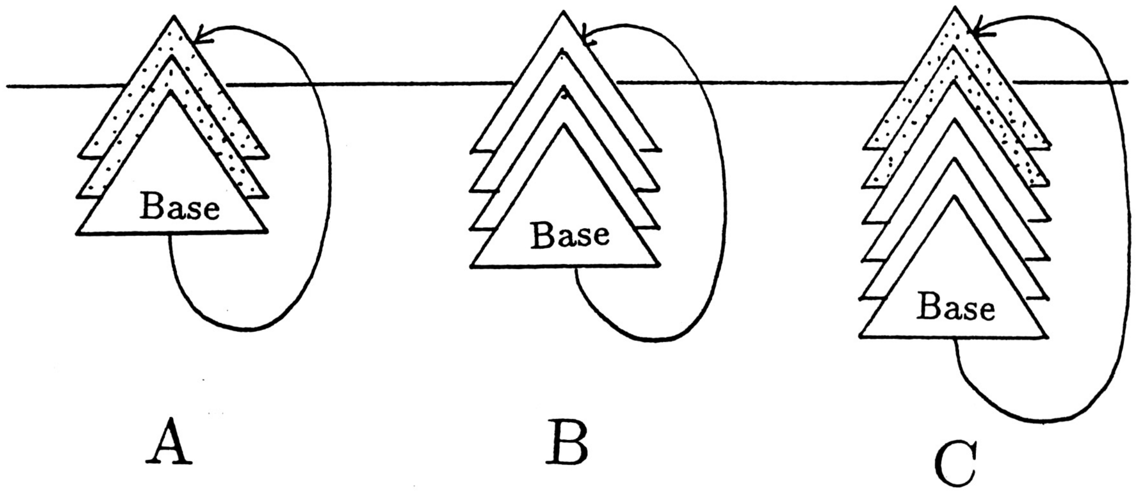

Figure 8: multiple inheritance

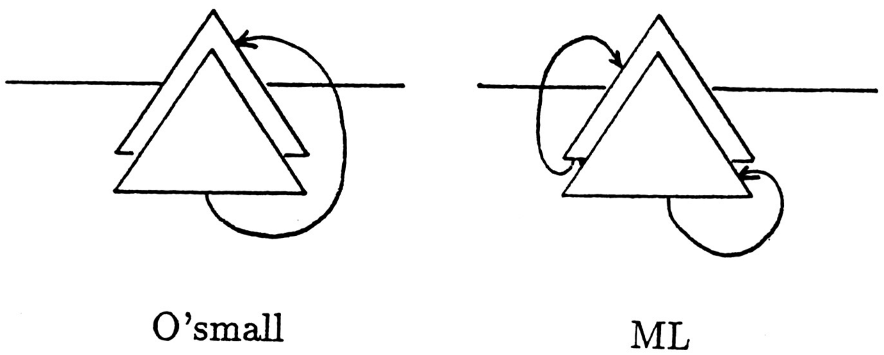

Figure 9: dynamic binding / static binding

Explicit use of wrappers bears a certain resemblance to functors in the module system of ML [Mac85]. Classes correspond to ML-structures. In contrast to wrappers, functors can have more than one argument and arguments must be qualified. The main difference is the absence of dynamic 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 figure 9. ML has no pseudo-variable *self* but it has recursion. Thus the arrows in the diagram stand for recursion in the functions of the module. A function of the old (lower) module will always recursively refer to functions of the old module, even if functions of the same name have been defined in the new module.

3 Conclusion

A new language construct and its denotational semantics have been presented. The resulting language is an extension of O'small [Hen90]. With explicit wrappers we are able to realize enhanced reusability of definitions, something that comes close to hierarchy inheritance, and a disciplined version of multiple inheritance. Our version of multiple inheritance can be described with the same mechanism as single inheritance.

Wrappers are not applicable to all classes but to a subset only. A static type system could check if a wrapper application is legal.

Acknowledgements
I would like to thank Gilad Bracha, Andreas Gündel, Reinhold Heckmann, and Reinhard Wilhelm for instructive discussions and comments on earlier drafts.

References

[BKK*86] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel, and Xerox Parc. CommonLoops: merging lisp and object-oriented programming. In *Object-Oriented Programming Systems, Languages and Applications*, pages 17–29, ACM, September 1986.

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

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

[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.

[Dij89] E. W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1397–1414, 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.

[Gun90] Andreas Gündel. Compatibility conditions on subclasses. 1990. unpublished notes.

[Hen90] Andreas V. Hense. *Wrapper Semantics of an Object Oriented Programming Language with State.* Technical Report A 14/90, Universität des Saarlandes, Fachbereich 14, 1990.

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

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

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

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

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

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

[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.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Object-Oriented Programming Systems, Languages and Applications*, ACM, Portland, Oregon, 1986.

[Sny85] Alan Snyder. *Object-Oriented Programming for Common Lisp*. Technical Report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985.

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

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

[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.

[Wol88] Mario Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, Manchester University, 1988.

[WZ88] P. Wegner and S. Zdonik. 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.

A Semantics of O'small

In this section we show how to extend the semantics of inheritance without state of section 1 to a semantics of an object oriented programming language, i.e. we add the state that was abstracted from in section 1. The proper semantics definition consists of the abstract syntax (section A.3) and the mapping from the syntactic domains onto the semantic domains defined by the semantic clauses (section A.4). Our way of describing semantics goes back to [Sto77] and [Gor79].

A.1 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 and wrappers. Expressible values are the same as denotable values. We will refer to them as denotable values from now on.

A.2 The New Semantic Domains

The newly introduced semantic domains are *Object*, *Class*, and *Wrapper*. Objects, classes, and wrappers were introduced in section 1. Their domains were:

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

What are the new semantic domains in the semantics of O'small which correspond to the semantics of inheritance? For *Object* it is still *Record*, although the record values are different now because the state is hidden inside the record values. The domain of wrappers is completely determined by the domain of classes and the latter is discussed now.

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 an existing class. The store remains unchanged because the instance variables are not allocated at the time of the class declaration 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

$$\alpha \rightarrow \alpha$$

where α is any domain (the domain of classes was *Object* $\rightarrow$ *Object* in section 1). The environment for methods is recursive whereas the environment for instance variables is not. We allocate the instance variables after the application of the fixed point operator. A function is needed for the allocation of all instance variables[4] of the new object. This function has to "know" the current store and has to return it with the instance variables inside it; the store must thus appear in the domain and the codomain of the function. In addition this function has to return an object. Therefore the result of the application of the fixed point operator to the class is:

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

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

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

A.3 Syntax of O'small

A.3.1 Syntactic Domains

There are primitive syntactic domains:

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

To the right are the variables ranging over the respective domain in the clauses. And there are 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
WrCl	the domain of wrapper and class declarations	W
Meth	the domain of method declarations	M

[4]including the instance variables declared in superclasses

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.

A.3.2 Syntactic Clauses

P	::= W K C
W	::= wrapper I def V in M | class I_1 = I_2 I_3 | W_1 W_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

Wrapper, class, variable, and method declarations may be empty.

A.4 Semantics of O'small

A.4.1 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:

$\text{Record}_{\alpha,\beta}$	$= \alpha \to [\beta + \{\perp\}]$	records	
Env	$= \text{Record}_{Ide,Dv}$	environments	r
Object	$= \text{Record}_{Ide,Dv}$	objects	o
Dv	$= \text{Loc} + \text{Rv} + \text{Method}_n + \text{Class} + \text{Wrapper}$	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. Only a finite subset of labels is mapped to values other than $\perp$.

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.

A.4.2 Semantic Clauses

The following semantic functions are primitive:

$$\begin{array}{ll} \mathsf{B}: & \text{Bas} \to \text{Bv} \\ \mathsf{O}: & \text{BinOp} \to \text{Rv} \to \text{Rv} \to \text{Store} \to [\text{Dv}\times\text{Store}] \end{array}$$

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:

$$\begin{array}{ll} \mathsf{P} & : \text{Pro} \to \text{File} \to \text{Ans} \\ \mathsf{R},\,\mathsf{E} & : \text{Exp} \to \text{Env} \to \text{Store} \to [\text{Dv}\times\text{Store}] \\ \mathsf{C} & : \text{CExp} \to \text{Env} \to \text{Store} \to [\text{Dv}\times\text{Store}] \\ \mathsf{V} & : \text{Var} \to \text{Env} \to \text{Store} \to [\text{Env}\times\text{Store}] \\ \mathsf{W} & : \text{WrCl} \to \text{Env} \to \text{Store} \to [\text{Env}\times\text{Store}] \\ \mathsf{M} & : \text{Method} \to \text{Env} \to \text{Env} \end{array}$$

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. For the definition of auxiliary functions in the following clauses refer to appendix B.

$$\begin{array}{l} \mathsf{P}[\![\text{W C}]\!]\ \text{i} = \text{extractans } \text{s}_{final} \\ \quad \textbf{where} \\ \quad \text{extractans} = \lambda\text{s}.(\text{s out}, \left\{\begin{array}{l}\text{error, if s err} \\ \text{stop , otherwise}\end{array}\right\}) \\ \quad (\text{r}_{wrcl},\_) = \mathsf{W}[\![\text{W}]\!]\ \text{r}_{initial}\ \text{s}_{initial} \\ \quad (\_,\text{s}_{final}) = \mathsf{C}[\![\text{C}]\!]\ \text{r}_{wrcl}\ \text{s}_{initial} \\ \quad \text{r}_{initial} = \left[\ \text{Base} \ \mapsto\ \lambda\text{o}.\lambda\text{s.result } []\ \right] \\ \\ \quad \text{s}_{initial} = \left[\begin{array}{lll}\text{err} & \mapsto & \text{false} \\ \text{inp} & \mapsto & \text{i} \\ \text{out} & \mapsto & \epsilon\end{array}\right] \end{array}$$

An answer from a program is gained by running it with an input. The store is initialized with the error flag set to *false* , the input, and an empty output. The initial environment contains 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 definitions of $.\star.$ refer to the appendix B.

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

The semantic function R produces R-values.

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

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 selector *I*. This should result in a method that is then applied to the parameters.

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

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*.

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

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

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].

$$\mathsf{W}\ [\![\text{wrapper } I \text{ def } V \text{ in } M]\!]\ r = \text{result } [I \mapsto w]$$

$$\textbf{where } w = \lambda x_{self}.\lambda x_{super}.\lambda s_{create}.\ (\mathsf{M}[\![M]\!] \left(\left[\begin{array}{lll} \text{self} & \mapsto & r_{self} \\ \text{super} & \mapsto & r_{super} \end{array} \right] \oplus r_{local} \oplus r \right), s_{new})$$

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

The wrapper w is bound to an identifier. w takes a class for self reference, a class 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. This is the reason for encapsulated instance variables.

In the inner where-clause above, environments and stores are created "successively". The instance variables for the superclass are allocated first. The instance variables for the current class are allocated by evaluating the declarations V in the changed store. Then x_{self} is again applied to the changed store to give the self-environment. The careful reader may have noticed that the resulting store is not needed. This is indicated by an 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.

$$\mathsf{W}\,[\![\text{class } \mathrm{I}_1 = \mathrm{I}_2\ \mathrm{I}_3]\!]\ \mathrm{r} = \mathsf{E}[\![\mathrm{I}_3]\!]\ \mathrm{r}\ \star \text{ Class? } \star\ \lambda \mathrm{c}.\ \mathsf{E}[\![\mathrm{I}_2]\!]\ \mathrm{r}\ \star \text{ Wrapper? } \star\ \lambda \mathrm{w}.\ \text{result}[\ \mathrm{I}_1 \mapsto \mathrm{w}\ \boxed{\triangleright}\ \mathrm{c}\ ]$$

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. The store remains unchanged when a class is declared.

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

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

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

B 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:

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

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

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

and is defined by

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

$\star$ is left associative. The definition of $\triangleright$ in section 1 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:

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

where $\oplus_{lpr}$ stands for the operation on records that is defined in definition 1.2. This is the only change of the inheritance function (definition 1.4). Here are further auxiliary functions. Let D be any semantic domain:

cond : [D $\times$ D] $\rightarrow$ Bool $\rightarrow$ D .... Alternative
cond(d_1, d_2) = λb.b $\rightarrow$ d_1, d_2

cnt : Loc $\rightarrow$ Store $\rightarrow$ [[Sv + {$\bot$}] $\times$ Store] .... Contents of a location
cnt = λl.λs.(s l, s)

cont : Dv $\rightarrow$ Store $\rightarrow$ [Sv $\times$ Store] .... Contents of a location with domain checking
cont = Loc? $\star$ cnt $\star$ Sv?

D? : D' $\rightarrow$ Store $\rightarrow$ [D' $\times$ Store], with D $\subseteq$ D' .... Domain checking

$$\text{D?} = \lambda\text{d.} \left\{ \begin{array}{l} \text{result d, if isD d} \\ \text{seterr , otherwise} \end{array} \right.$$

deref : Dv $\rightarrow$ Store $\rightarrow$ [Dv $\times$ Store] .... Dereferencing

$$\text{deref} = \lambda\text{e.} \left\{ \begin{array}{l} \text{cont e , if isLoc e} \\ \text{result e, otherwise} \end{array} \right.$$

new : Store $\rightarrow$ [Loc $\times$ Store] .... Getting a new location in the store
new s = (l,s) or = ($\bot$, [err $\mapsto$ true] $\oplus$ s)
If new s = (l,s) then s l = $\bot$ is guaranteed.

result : D $\rightarrow$ Store $\rightarrow$ [D $\times$ Store] .... Side effect free evaluation
result d = λs.(d, s)

seterr : Store $\rightarrow$ [D $\times$ Store] .... Setting the error flag
seterr = λs.($\bot$, [err $\mapsto$ true] $\oplus$ s)

update : Loc $\rightarrow$ Dv $\rightarrow$ Store $\rightarrow$ [Dv$\times$Store] .... Updating of a location
update l = Sv? $\star$ λe.λs.(unit, [l $\mapsto$ e] $\oplus$ s)