Class-based versus Object-based: A Denotational Comparison

Bernhard Reus*

School of Cognitive and Computing Sciences
University of Sussex
bernhard@sussex.ac.uk

Abstract. In object-oriented programming one distinguishes two kinds of languages. Class-based languages are centered around the concept of classes as descriptions of objects. In object-based languages the concept of a class is substituted by constructs for the creation of individual objects. Usually, the object-based languages attract interest because of their "simplicity". This paper contains a thorough denotational analysis which reveals that simplicity is quickly lost if one tackles verification issues. This is due to what is sometimes called "recursion through the store". By providing a denotational semantics for a simple class-based and a simple object-based language it is shown that the denotational semantics of the object-based language needs much more advanced domain theoretic machinery than the class based one. The gap becomes even wider when we define concepts of specification and appropriate verification rules.

1 Introduction and Motivation

In their seminal book *A Theory of Objects* [1, Chapter 4] Abadi and Cardelli write "*Object-based languages are intended to be both simpler and more flexible than traditional class-based languages.*" and "*Object-based languages are refreshing in their simplicity, compared with even simple class based languages.*" While object-based languages are definitely *more flexible*, they are *simpler* only with respect to syntax but much more involved with respect to semantics and programming logics. To demonstrate this is the main objective of this paper.

Semantics in object-oriented languages usually refers to some operational semantics. Also [1] mainly deal with syntax, operational semantics and typing. Denotational semantics does not get much attention although e.g. *loc.cit.* [Chap. 14] contains a sophisticated typed denotational semantics for a functional object calculus. Consequently, programming logics (axiomatic semantics) for object-oriented languages are usually treated in a syntactic style, proving soundness of a logic w.r.t. the operational semantics of a language [2, 4, 8, 15, 20]. Some notable exceptions are the extension of "higher-order Algol" to objects [16, 11] and the co-algebraic approach propagated by [10]. The former uses

* partially supported by the EPSRC under grant GR/R65190/01 and by the Nuffield Foundation under grant NAL/00244/A

a different style which makes comparison with object-based languages difficult and the latter does not give rise to any particular proof principles apart from co-induction which does not seem appropriate for the object-based case (see [19]).

Denotational techniques were quite successfully applied to the verification of functional programs in the LCF project (Logic of Computable Functions [13]). It is more abstract and allows for concise soundness proofs. Therefore, we use denotational semantics for a rigorous analysis of class-based and object-based languages.

Class-based languages use the *class* concept to describe a collection of objects that are the *instances* of the class. Object-based languages get around the notion of *class* by providing features for the creation of individual objects. Classes separate methods from objects but the class types still establish a link. They indicate to which collection an object belongs and which method suite applies. In the object-based case any object provides its own method suite. Method invocation is therefore always with respect to an object's methods the code of which is stored in heap. Recursive methods can be simply obtained by calling the same method of the same object. Object specifications must account for this kind of recursion and necessarily become recursive themselves. Therefore, existence of specifications cannot be taken for granted. We will use the techniques presented in [14] to obtain an existence result.

As representatives for object-based and class-based languages we chose the imperative untyped object-calculus of [1] and a very rudimentary sublanguage of Java, respectively. The syntax of both languages is about equally rich. Throughout the paper we just deal with an untyped semantics[1]. As types are properties of syntax they can be expressed on the specification level.

The results of the object-based languages appear in greater detail in [19], the class-based analysis and the comparison are the original contributions. Some preliminary attempts have been published in [18, 17].

In [5] a different but related comparison has been carried out for various interpretations of the functional object calculus in type theory. It stops short of classes and does not discuss the imperative paradigm nor verification issues.

The paper is organised as follows: In Sect. 2 we introduce the syntax of both languages, the class-based and the object-based one. Their denotational semantics is given in the following section. Programming logics on the level of denotations are then discussed, again for both languages, in Section 4 by providing a notion of specification and discussing verification rules. A comparison in a nutshell (Sect. 5) and some remarks about further extensions (Sect. 6) conclude the paper.

[1] This does not mean that we ignore types completely. In the class-based case the type of an object is essential for method dispatch.

2 Syntax

In this section the syntax of the class-based and object-based sample languages is introduced. We restrict our attention to small kernel languages.

2.1 Preliminaries

The name space is divided into class names $\mathcal{C}$, field names $\mathcal{F}$, and method names $\mathcal{M}$. Moreover, Var denotes a infinitely countable set of untyped variables.

Note that in the rest of the paper $x \in \mathsf{Var}$ will always denote a variable and all (possibly qualified) appearances of m will refer to labels in $\mathcal{M}$, all (possibly qualified) appearances of f will refer to labels in $\mathcal{F}$ and all (possibly qualified) appearances of C will refer to labels in $\mathcal{C}$.

2.2 Class-based

The syntax of our simple class-based language is defined below. For the sake of simplicity, we omit method parameters, access modes, static attributes and static methods.

$$
\begin{array}{rlll}
a, b & ::= & x & \text{variables} \\
 & \mid & \texttt{new}\ \mathsf{C}() & \text{object creation} \\
 & \mid & a.\mathsf{f} & \text{field selection} \\
 & \mid & a.\mathsf{f} = b & \text{field update} \\
 & \mid & a.\mathsf{m}() & \text{method call} \\
 & \mid & \texttt{let}\, x{=}a\ \texttt{in}\, b & \text{local def.}
\end{array}
$$

The "self" reference `this` is just considered a special (predefined) variable name, therefore $\texttt{this} \in \mathsf{Var}$. Note that we do not distinguish between expressions and statements (like in the object calculus). There is no extra operator for sequential composition as it can be expressed using "let", i.e. $a; b \equiv \texttt{let}\ x = a\ \texttt{in}\ b$ where x does not occur freely in a and b. Methods are considered non-void but this is not a real restriction.

Additionally, there is a syntax for class definitions:

$$
\begin{array}{rll}
c & ::= & \texttt{class}\ \mathsf{C}\ \{\ \mathsf{f}_i\ ^{i=1,\dots,n} \\
 & & \qquad\qquad \mathsf{m}_j = b_j\ ^{j=1,\dots,m} \\
 & & \qquad\qquad \mathsf{C}\,() = b \\
 & & \qquad\qquad \} \\
cl & ::= & \epsilon \mid c\ cl
\end{array}
$$

A class definition is similar to one in Java with all type declarations omitted. A class definition contains a list of field declarations, a list of method declarations and a constructor. All functions and the constructor are nullary.

This syntax is similar to the one of Featherweight Java [9] or Java-light [12] but the former is purely functional and the latter is typed and covers a larger subset of the Java syntax.

2.3 Object-based

Recall that the syntax of the imperative untyped object calculus of [1] is as follows:

$$
\begin{array}{rll}
a, b ::= & x & \text{variable} \\
\mid & [\mathsf{f}_i{=}b_i \ ^{i=1..n}, \mathsf{m}_j{=}\varsigma(x_j)b_j \ ^{j=1..m}] & \text{object creation} \\
\mid & a.\mathsf{f} & \text{field select} \\
\mid & a.\mathsf{f}{\Leftarrow}b & \text{field update} \\
\mid & a.\mathsf{m}() & \text{method call} \\
\mid & a.\mathsf{m}{\Leftarrow}\varsigma(x)b & \text{method update} \\
\mid & \mathtt{clone}(a) & \text{shallow copy} \\
\mid & \mathtt{let}\, x{=}a \,\mathtt{in}\, b & \text{local def.}
\end{array}
$$

Note that, by contrast to [1], we do distinguish between fields and methods explicitly.

3 Denotational Semantics

In this section we give a denotational semantics for both languages, the class-based and the object-based one.

3.1 Preliminaries

When specifying the recursive types needed for the interpretation of object calculi we often have to employ record type formation in the following sense. Let $\mathbb{L}$ be a (countable) set of labels and A a predomain then the type of records with entries from A and labels from $\mathbb{L}$ is defined as follows:

$$\mathsf{Rec}_{\mathbb{L}}(A) = \Sigma_{L \in \mathcal{P}_{\mathsf{fin}}(\mathbb{L})} A^L$$

where A^L is the set of all total functions from L to A. It can be easily seen that $\mathsf{Rec}_{\mathbb{L}}$ is a locally continuous functor on predomains with continuous maps. A record with labels l_i and corresponding entries v_i $(1 \le i \le k)$ is written $\{\!| l_1 = v_1, \dots l_k = v_k |\!\}$. Notice that $\mathsf{Rec}_{\mathbb{L}}(A)$ is always non–empty as it contains the element $\langle \emptyset, \emptyset \rangle$.

Definition 1. *The update (and extension) operation for records is defined as in Table 1. If $r \in \mathsf{Rec}_{\mathbb{L}}(A)$ and $l \in \mathbb{L}$ we will write $r.l{\downarrow}$ to express that l is in the domain of r, ie. label l is defined in r.*

Some other basic domains are needed. Let Loc be some countable set of locations (addresses) for objects and Val the set of basic untyped values like booleans or integers. Observe that $\mathsf{Loc} \subseteq \mathsf{Val}$, ie. locations can be seen as values and thus be stored in environments. Let $\mathsf{Env} = \mathsf{Loc}^{\mathsf{Var}}$ denote the domain of local environments (or stacks) that are used to deal with bound variables.

$$\{\!|\mathsf{m}_i{=}f_i|\!\}^{i=1\ldots n}[\mathsf{m}{:=}f] = \begin{cases} \{\!|\mathsf{m}_1{=}f_1,\ldots,\mathsf{m}_n{=}f_n,\mathsf{m}{=}f|\!\} & \text{if } \mathsf{m} \text{ is different from all } \mathsf{m}_i \\ \{\!|\mathsf{m}_1{=}f_1,\ldots,\mathsf{m}{=}f,\ldots,\mathsf{m}_n{=}f_n|\!\} & \text{if } \mathsf{m} = \mathsf{m}_i \end{cases}$$

Table 1. Record update and extension

It is sensible to assume that Val is a flat predomain, ie. any ascending chains in Val consist of one element only. In the same way, Loc is a flat predomain too.

In general, we use predomains (domains without a least element) as they are simpler to deal with. Note that the category of predomains and continuous partial maps ($\rightharpoonup$) is equivalent to the category of domains with strict continuous maps. As usual, $\times$ binds stronger than $\rightharpoonup$.

As the concept of a "method" is important to class-based as well as object-based languages let us define a domain Cl (for *closure*) in which methods (always without arguments in this paper) will find their interpretation:

$$\mathsf{Cl} = \mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}$$

A closure is a partial function mapping a location, representing the callee object, and (the old) store to a value, the result of the method, and a new store which accounts for the side effects during execution of the method in question.

3.2 Class-based

The class-based language introduced above finds its interpretation within the following system of predomains:

Definition 2. *Let the semantic counterparts of objects (1), stores (2), and method suites (3) be defined as below.*

(1) $\mathsf{Ob} = \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathcal{C}$
(2) $\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$
(3) $\mathsf{Ms} = \mathsf{Rec}_{\mathcal{C}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \times (\mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St}))$

Note that an object consists of a record of field values plus the name of its class type. This information is stored in order to be able to do the dynamic dispatch for method calls (due to subtype polymorphism). A *method suite* is a record $\mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \times (\mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St})$ which contains all methods of a class including the constructor `new`. Therefore Ms is the record of all method suites for all declared classes. It should be stressed that the definition of St as $\mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$ faithfully reflects the idea of a state as an assignment of objects to a finite set of locations. Moreover, this formulation, by contrast to the functional modelling of states, has the advantage that St is a *flat* predomain.

The function $\mathsf{type} : \mathsf{Ob} \to \mathcal{C}$ that returns an object's type, ie. $\mathsf{type}(o) = o.2$ will be useful later.

The interpretation of the syntax is given by the following semantic equations:

Definition 3. *Given an environment* $\rho \in \mathsf{Env}$*, an environment of method suites* $\mu \in \mathsf{Ms}$ *and an expression* a *its interpretation* $[\![a]\!]\rho : \mathsf{Ms} \rightharpoonup \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}$ *is defined in Table 2 (page 6). Note that the semantic* ***let*** *on the right hand side of the definitions is strict in its first argument.*

$$
\begin{array}{ll}
[\![x]\!]\,\rho\,\mu\,\sigma & = \langle \rho(x), \sigma\rangle \\
[\![\mathtt{this}]\!]\,\rho\,\mu\,\sigma & = \langle \rho(\text{this}), \sigma\rangle \\
[\![\mathtt{new}\ \mathsf{C}()]\!]\,\rho\,\mu\,\sigma & = \mu.\mathsf{C}.2(\sigma) \\
[\![a.\mathsf{f}]\!]\,\rho\,\mu\,\sigma & = \textbf{let}\ \langle \ell, \sigma'\rangle{=}[\![a]\!]\,\rho\,\mu\,\sigma\ \textbf{in}\ \sigma'.\ell.1.\mathsf{f} \\
[\![a.\mathsf{f}{=}b]\!]\,\rho\,\mu\,\sigma & = \textbf{let}\ \langle \ell, \sigma'\rangle{=}[\![a]\!]\,\rho\,\mu\,\sigma\ \textbf{in} \\
 & \quad\ \textbf{let}\ \langle v, \sigma''\rangle{=}[\![b]\!]\,\rho\,\mu\,\sigma'\ \textbf{in}\ \langle v, \sigma''[\ell := \langle \sigma''.\ell.1[\mathsf{f} := v], \sigma''.\ell.2\rangle]\rangle \\
[\![a.\mathsf{m}()]\!]\,\rho\,\mu\,\sigma & = \textbf{let}\ \langle \ell, \sigma'\rangle{=}[\![a]\!]\,\rho\,\mu\,\sigma\ \textbf{in}\ \mu.\mathsf{type}(\sigma'.\ell).1.\mathsf{m}(\langle \ell, \sigma'\rangle) \\
[\![\mathtt{let}\ x{=}a\ \mathtt{in}\ b]\!]\,\rho\,\mu\,\sigma & = \textbf{let}\ \langle \ell, \sigma'\rangle{=}[\![a]\!]\,\rho\,\mu\,\sigma\ \textbf{in}\ [\![b]\!]\,\rho[\ell/x]\,\mu\,\sigma'\ .
\end{array}
$$

Table 2. Denotational semantics of expressions (class-based)

Given an environment of method suites $\mu \in \mathsf{Ms}$ *a class definition* c *finds its interpretation* $[\![c]\!] : \mathsf{Ms} \rightharpoonup \mathsf{Ms}$ *as defined in Table 3 (page 6). Accordingly, the semantics of a list of class definitions is as follows:*

$$
\begin{array}{ll}
[\![c\ cl]\!]\,\mu & = [\![cl]\!]\,\mu[\mathsf{C}{:=}[\![c]\!]\,\mu] \\
[\![\epsilon]\!]\,\mu & = \mu
\end{array}
$$

As this does not allow one to write a recursive class definition the right semantics $[\![cl]\!] : \mathsf{Ms}$ *of a class definition list is as follows*

$$[\![cl]\!] = \text{fix}\ \lambda\mu.\,[\![cl]\!]\,\mu$$

where the usage of the fixpoint operator is justified since Ms *is a domain and* $[\![cl]\!] : \mathsf{Ms} \rightharpoonup \mathsf{Ms}$ *is a continuous function between domains. Continuity is a consequence of the definitions in Tables 2 and 3. All interpretations depending on the method suites (method application and object creation) are continuous in* μ*.*

$$
\left[\!\!\left[
\begin{array}{l}
\mathtt{class}\ \mathsf{C}\ \{\ \mathsf{f}_i{}^{\,i=1..n} \\
\qquad\quad \mathsf{m}_j{=}b_j{}^{\,j=1..m} \\
\qquad\quad \mathtt{C}()\ {=}b \\
\qquad\ \ \}
\end{array}
\right]\!\!\right]
\mu = \left\langle
\begin{array}{l}
\{\!|\mathsf{m}_j{=}\lambda\langle\ell,\sigma\rangle.\,[\![b_j]\!]\,\rho[\ell/\text{this}]\,\mu\,\sigma|\!\}^{j=1..m}, \\
\lambda\sigma.\,\langle \ell, [\![b]\!]\,\rho[\ell/\text{this}]\,\mu\,\sigma[\ell{:=}\langle\{\!|\mathsf{f}_i{=}\text{init}_{\mathsf{C},i}|\!\}^{i=1..n}, \mathsf{C}\rangle]\rangle
\end{array}
\right\rangle
$$

where ℓ is a fresh location not in the domain of σ and $\text{init}_{\mathsf{C},i}$ denotes a default value for field f_i in C.

Table 3. Denotational semantics of class definitions

$$
\begin{array}{ll}
[\![x]\!]\rho\,\sigma & = \langle \rho(x), \sigma\rangle \\
\left[\!\!\left[\begin{matrix}[\mathsf{f}_i{=}\varsigma(x_i)b_i\ ^{i=1..n}, \\ \mathsf{m}_j{=}\varsigma(x_j)b_j\ ^{j=1..m}]\end{matrix}\right]\!\!\right] & = \left\langle \ell, \begin{matrix}\sigma[\ell := \langle\ \{\!|\mathsf{f}_i{=}[\![b_i]\!]\,\rho\,\sigma|\!\}^{i=1..n}, \\ \qquad\qquad \{\!|\mathsf{m}_j{=}\lambda\langle\ell',\sigma'\rangle.\ [\![b_j]\!]\,\rho[\ell'/x_j]\,\sigma'|\!\}^{j=1..m}\ \rangle]\end{matrix}\right\rangle \\
 & \quad \text{where } \ell \text{ is a fresh location not in the domain of } \sigma \\
[\![a.\mathsf{f}]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in}\ \sigma'.\ell.1.\mathsf{f} \\
[\![a\,.\,\mathsf{f}{\Leftarrow}b]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in} \\
 & \qquad \langle\ell,\sigma'[\ell{:=}\langle\sigma'.\ell.1[\mathsf{f}{:=}[\![b]\!]\rho\,\sigma'],\sigma'.\ell.2\rangle]\rangle \\
[\![a.\mathsf{m}()]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in}\ \sigma'.\ell.2.\mathsf{m}(\langle\ell,\sigma'\rangle) \\
[\![a\,.\,\mathsf{m}{\Leftarrow}\varsigma(x)b]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in} \\
 & \qquad \langle\ell,\sigma'[\ell{:=}\langle\sigma'.\ell.1,\sigma'.\ell.2[\mathsf{m}{:=}\lambda\langle\ell',\sigma''\rangle.[\![b]\!]\,\rho[\ell'/x]\,\sigma'']\rangle]\rangle \\
[\![\mathtt{clone}(a)]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in}\ \langle\ell',\sigma'[\ell'{:=}\sigma'.\ell]\rangle \\
 & \quad \text{where } \ell' \text{ is a fresh location not in the domain of } \sigma' \\
[\![\mathtt{let}\,x{=}a\ \mathtt{in}\,b]\!]\rho\,\sigma & = \mathbf{let}\ \langle\ell,\sigma'\rangle{=}[\![a]\!]\rho\,\sigma\ \mathbf{in}\ [\![b]\!]\rho[\ell/x]\,\sigma'\ .
\end{array}
$$

Table 4. Denotational semantics of the imperative object calculus

3.3 Object-based

The imperative object calculus finds its interpretation within the following system of *recursive* types:

Definition 4. *Let the semantic counterparts of objects (1) and stores (2) be defined as below:*

(1) $\mathsf{Ob} = \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl})$
(2) $\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$

In the following we will write $\mathsf{St}_{\mathsf{Val}}$ *for* $\mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}))$*. There is an obvious projection* $\pi_{\mathsf{Val}} : \mathsf{St} \to \mathsf{St}_{\mathsf{Val}}$ *with* $\pi_{\mathsf{Val}}(\sigma).\ell \simeq \pi_1(\sigma.\ell)$ *where* π_1 *projects on the first component.*

The interpretation of the syntax is given by the following semantic equations:

Definition 5. *Given an environment* $\rho \in \mathsf{Env} = \mathsf{Loc}^{\mathsf{Var}}$ *and an object expression* a *its interpretation* $[\![a]\!]\rho : \mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St}$ *is defined in Table 4 (page 7).*

In the object-based case the semantics of field and method select are identical. So are the semantics of field and method update. The only difference is that the fields are of the simpler type Val.

Note that we can still write recursive methods although semantically there is no fixpoint of recursive definitions as for the class-based case. The recursion works simply "through the store", it is established by self application. For example, if $[\![[\mathtt{m} = \varsigma(x)x\mathtt{.m()}]]\!]\ \rho\ \sigma = \langle \ell, \sigma^n\rangle$ where

$$\sigma^n := \sigma[\ell := \langle \emptyset, \{\!|\mathsf{m} = \lambda\langle\ell',\sigma'\rangle.\ \sigma'.\ell'2.\mathsf{m}(\langle\ell',\sigma'\rangle)|\!\}\rangle]$$

we obtain

$$\begin{aligned}
[\![[\mathtt{m} = \varsigma(x)x\mathtt{.m()}]\mathtt{.m()}]\!]\ \rho\ \sigma &= \mathbf{let}\ \langle \ell, \sigma' \rangle = [\![[\mathtt{m} = \varsigma(x)x\mathtt{.m()}]]\!]\ \rho\ \sigma\ \mathbf{in} \\
&\qquad \sigma'.\ell.2.\mathsf{m}(\langle \ell, \sigma' \rangle) \\
&= \sigma^n.\ell.2.\mathsf{m}(\langle \ell, \sigma^n \rangle) \\
&= \sigma^n.\ell.2.\mathsf{m}(\langle \ell, \sigma^n \rangle) \\
&= \ldots
\end{aligned}$$

which does not yield a defined result.

4 Logics of Programs

Once the denotational semantics of classes and objects is settled one can start reasoning about these denotations. This is in analogy with the LCF (Logic of Computable Functions) project (see e.g. [13]) for the functional paradigm. In LCF one reasons about functional programs by referring to their semantic (fixpoint) interpretation using results of domain theory. An analogous procedure will be carried out for the object-oriented languages defined above. Reasoning about classes and objects will be done by referring to their semantics defined in the previous section.

4.1 Specifications

First we have to define an appropriate notion of *specification* for classes and objects. For the class-based case we follow [8, 15, 20], for the object-based case [2]. Whereas in the cited approaches specifications are defined syntactically and their semantics implicitly by proof rules (verified w.r.t. some operational semantics) we work consistently on the denotational level. The more abstract description leads to a conceptually clearer understanding of the principles involved and to better structured soundness proofs.

As every class and every object in the object-based language contain a number of methods, some common structures in specifications can be singled out: for every method there is a result specification and a transition specification. This distinction is usually blurred in class-based verification calculi, but we will see below why it is important for object-based specifications.

$B_{\mathsf{m}} \in \wp(\mathsf{Val} \times \mathsf{St})$	result specification	for $\mathsf{m} \in \mathcal{M}$
$T_{\mathsf{m}} \in \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St})$	transition specification	for $\mathsf{m} \in \mathcal{M}$

Taking into account that we are only interested in partial correctness (at least for the considerations of this paper) the meaning of the specifications can be described informally as follows:

B_{m} : "If method m terminates its result fulfils the result specification."
T_{m} : "If method m terminates its effect fulfils the transition specification."

An object is given by a location and the store it resides in and a class by its name and the method suites it defines. Therefore, a class specification is a predicate in $\wp(\mathcal{C} \times \mathsf{Ms})$ whereas an object specification is a predicate in $\wp(\mathsf{Loc} \times \mathsf{St})$.

4.2 Class-based

Definition 6. *Specifications of methods,* Spec_C^m*, classes* Spec_C*, and packages,* $\mathbf{Spec}_C$*, are defined below:*

$$
\begin{array}{rl}
\mathrm{Spec}_C^m & : \wp(\mathsf{Val} \times \mathsf{St}) \times \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}) \;\to\; \wp(\mathcal{C} \times \mathsf{Cl}) \\
\mathrm{Spec}_C & : \mathsf{Rec}_{\mathcal{M}}(\wp(\mathsf{Val} \times \mathsf{St}) \times \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St})) \;\to\; \wp(\mathcal{C} \times \mathsf{Ms}) \\
\mathbf{Spec}_C & : \mathsf{Rec}_{\mathcal{C}}(\mathsf{Rec}_{\mathcal{M}}(\wp(\mathsf{Val} \times \mathsf{St}) \times \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}))) \;\to\; \wp(\mathsf{Ms})
\end{array}
$$

$$
\begin{array}{l}
\langle \mathsf{C}, f \rangle \in \mathrm{Spec}_C^m(B_{\mathsf{m}}, T_{\mathsf{m}}) \quad \textit{iff} \quad \forall \ell \in \mathsf{Loc}.\, \forall v \in \mathsf{Val}.\, \forall \sigma, \sigma' \in \mathsf{St}. \\
\qquad\qquad (\mathsf{type}(\sigma.\ell) = \mathsf{C} \wedge f(\ell, \sigma) = \langle v, \sigma' \rangle) \Rightarrow B_{\mathsf{m}}(v, \sigma') \wedge T_{\mathsf{m}}(\ell, \sigma, v, \sigma')
\end{array}
$$

$$
\begin{array}{l}
\langle \mathsf{C}, \mu \rangle \in \mathrm{Spec}_C(r) \quad \textit{iff} \quad \forall \mathsf{m} \in \mathcal{M}. \\
\qquad\qquad r.\mathsf{C}.\mathsf{m}{\downarrow} \Rightarrow \mu.C.1.\mathsf{m}{\downarrow} \;\wedge\; \langle \mathsf{C}, \mu.\mathsf{C}.1.\mathsf{m} \rangle \in \mathrm{Spec}_C^m(r.\mathsf{C}.\mathsf{m})
\end{array}
$$

$$
\mu \in \mathbf{Spec}_C(R) \;\; \textit{iff} \;\; \forall \mathsf{C} \in \mathcal{C}.\, R.\mathsf{C}{\downarrow} \;\Rightarrow \mu.\mathsf{C}{\downarrow} \;\wedge\; \langle \mathsf{C}, \mu \rangle \in \mathrm{Spec}(R.\mathsf{C})
$$

Typical examples of transition specifications T_{m} are specifications à la Hoare:

$$
T_{\mathsf{m}}(\ell, \sigma, v, \sigma') \equiv P(\ell, \sigma) \Rightarrow Q(\ell, \sigma, v, \sigma')
$$

Note that P may refer to the old state σ as well as to the value of the self parameter `this`, ℓ. On the other hand, Q may refer to the self parameter ℓ, to the new state σ' after method execution and to the result v of the method. It may also refer to the pre-state σ in order to be able to talk about the "old values" of variables.[2]

Theorem 1. *The specifications* Spec_C^m, Spec_C*, and* $\mathbf{Spec}_C$ *always exist and are admissible predicates.*

Proof. Existence is trivial as the specifications are defined explicitly. This point is only emphasized to address the difference with respect to object specifications. Let us show admissibility for Spec_C^m, the other cases work analogously. First we rewrite Spec_C^m:

$$
\begin{array}{l}
\langle \mathsf{C}, f \rangle \in \mathrm{Spec}_C^m(B_{\mathsf{m}}, T_{\mathsf{m}}) \;\; \text{iff} \;\; \forall \ell \in \mathsf{Loc}.\, \forall v \in \mathsf{Val}.\, \forall \sigma, \sigma' \in \mathsf{St}. \\
\qquad\qquad (\mathsf{type}(\sigma.\ell) \neq \mathsf{C} \vee f(\ell, \sigma){\uparrow} \vee (B_{\mathsf{m}}(f(\ell, \sigma)) \wedge T_{\mathsf{m}}(\ell, \sigma, f(\ell, \sigma)))
\end{array}
$$

As Cl is a flat predomain admissibility must only be shown with respect to the second component f. Since admissible predicates are closed under universal quantification, disjunction, conjunction, and composition with continuous maps (and function application is continuous) it only remains to show that $\uparrow$, B_{m}, and T_{m} are admissible. The former is by definition, the latter are because they are predicates on a flat predomain.

In order to prove that a specification is true for a program (package) one has to apply fixpoint induction. The corresponding proof rule reads as follows:

$$
\frac{\mu \in \mathbf{Spec}_C(R) \Rightarrow [\![cl]\!]\mu \in \mathbf{Spec}_C(R)}{\mathrm{fix}\ \lambda\mu.\, [\![cl]\!]\, \mu \in \mathbf{Spec}_C(R)}
$$

[2] Alternatively, one can employ logical variables to talk about old values of variables.

which presupposes that the specification is admissible.

This is a denotational equivalent of the inductive proof rule presented e.g. in [21][3].

Which method is executed for a method application $o.\mathsf{m}()$ actually depends on the type of o: $\mathsf{type}(\llbracket o \rrbracket)$. For verification purposes this is problematic as the dynamic type of an object is not known at verification (ie. compile) time. In [20] a special method triple $\{P\}\mathtt{C} :: \mathtt{m}()\{Q\}$ was suggested with the intuitive meaning that all methods $\mathtt{m}$ in class C and all subclasses of C fulfil the specification in terms of pre- and post-conditions P and Q. Moreover, the following proof rule was proposed for any method $\mathtt{m}$ provided it does not depend on other methods.

$$(\dagger) \quad \frac{\forall C' \leq C.\, \{P\}\, C :: \mathtt{m}()\, \{Q\} \vdash \{P\}\, \mathrm{body}(C' :: \mathtt{m})\{Q\}}{\{P\}\, C :: \mathtt{m}()\, \{Q\}}$$

where $\mathrm{body}(C' :: \mathtt{m})$ denotes the body of the corresponding method. This rule can be verified w.r.t. the operational semantics by induction on the number of unfoldings of method $\mathtt{m}$. A less tedious soundness proof uses fixpoint induction and a slightly stronger notion of specification:

Definition 7. *Specifications of packages obeying behavioural subtyping:*

$$\mathbf{Spec}^{\mathrm{beh}}_{C} \;:\; \mathsf{Rec}_{\mathcal{C}}(\mathsf{Rec}_{\mathcal{M}}(\wp(\mathsf{Val} \times \mathsf{St}) \times \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}))) \;\rightarrow\; \wp(\mathsf{Ms})$$
$$\mu \in \mathbf{Spec}^{\mathrm{beh}}_{C}(R) \textit{ iff } \forall \mathsf{C} \in \mathcal{C}.\ \forall \mathsf{C}' \geq \mathsf{C}.\ R.\mathsf{C}'{\downarrow} \Rightarrow \langle \mathsf{C}, \mu\rangle \in \mathrm{Spec}_{C}(R.\mathsf{C}')$$

Thus, a package μ fulfils specification $\mathbf{Spec}^{\mathrm{beh}}_{C}(R)$ if all classes in μ are correct w.r.t. their own class definition but also the specifications of all super classes. By Theorem 1 and the fact that admissible predicates are closed under conjunction one obtains that $\mathrm{Spec}^{\mathrm{beh}}_{C}$ is admissible, too.

Theorem 2. *If the method* $\mathtt{m}$ *in class* $\mathtt{C}$ *is recursive but does not depend on other methods, then the rule* $(\dagger)$ *is sound w.r.t. our semantics.*

Proof. Let R be the specification $\mathbf{Spec}_{\mathcal{C}}(\{\!|\mathsf{C} = \{\!|\mathsf{m} = \langle \mathrm{True}, T_{\mathsf{m}}\rangle|\!\}|\!\})$ where $T_{\mathsf{m}}(\ell, \sigma, v, \sigma') = P(\ell, \sigma) \Rightarrow Q(\ell, v, \sigma')$. Then the semantic counterpart of the rule above is:

$$\frac{\forall \mathsf{C}' \leq \mathsf{C}.\, \mu \in \mathrm{Spec}^{\mathrm{beh}}_{C}(R) \Rightarrow (\llbracket cl \rrbracket \mu).\mathsf{C}'.1.\mathsf{m} \in \mathrm{Spec}^{m}_{C}(\mathrm{True}, T_{\mathsf{m}})}{\mathrm{fix}\lambda\mu.\, \llbracket cl \rrbracket\, \mu \in \mathrm{Spec}^{\mathrm{beh}}_{C}(R)}$$

which is equivalent to

$$\frac{\mu \in \mathrm{Spec}^{\mathrm{beh}}_{C}(R) \Rightarrow \forall \mathsf{C}' \leq \mathsf{C}.\, (\llbracket cl \rrbracket \mu).\mathsf{C}'.1.\mathsf{m} \in \mathrm{Spec}^{m}_{C}(\mathrm{True}, T_{\mathsf{m}})}{\mathrm{fix}\lambda\mu.\, \llbracket cl \rrbracket\, \mu \in \mathrm{Spec}^{\mathrm{beh}}_{C}(R)}$$

Under the given assumptions it is easy to show that

$$\forall \mathsf{C}' \leq \mathsf{C}.\, (\llbracket cl \rrbracket \mu).\mathsf{C}'.1.\mathsf{m} \in \mathrm{Spec}^{m}_{C}(\mathrm{True}, T_{\mathsf{m}}) \text{ iff } \llbracket cl \rrbracket \mu \in \mathbf{Spec}^{\mathrm{beh}}_{C}(R)$$

[3] More precisely, the semantical equivalent of [21] is rather computational induction which implies fixpoint induction.

which then is equivalent to the valid fixpoint induction rule

$$\frac{\mu \in \mathrm{Spec}_C^{\mathrm{beh}}(R) \Rightarrow [\![cl]\!]\mu \in \mathrm{Spec}_C^{\mathrm{beh}}(R)}{\mathrm{fix}\lambda\mu.\,[\![cl]\!]\,\mu \in \mathrm{Spec}_C^{\mathrm{beh}}(R)}$$

Apart from the method specifications T_{m}, class specifications could be extended to contain requirements on the constructor `new` (as well as on any static method or field).

Given method suites μ, a class invariant for C is a predicate on *objects* $I \in \wp(\mathsf{Loc} \times \mathsf{St})$ such that all methods $\mu.\mathsf{C.m}$ in class C fulfil the method specification

$$T_{\mathsf{m}}(\ell, \sigma, v, \sigma') \equiv I(\ell, \sigma) \Rightarrow I(\ell, \sigma')$$

and object creation establishes I, in other words $I(\mu.\mathsf{C}.2(\sigma))$ for all $\sigma \in \mathsf{St}$.

4.3 Object-based

Due to self application the argument of an object's method is the object itself. Proving properties of an object means proving properties of its methods but this requires that the object itself already satisfies the specification.

A first tentative definition of an object specification is given below. Note that we add an additional component $A \in \wp(\mathsf{Loc} \times \mathsf{St})$ to describe properties of the fields not covered by the method result and transition specifications.

Definition 8. *A tentative object specifications is as follows:*

$$\begin{aligned}\mathrm{Spec}_o \;:\; & (\wp(\mathsf{Loc} \times \mathsf{St}) \times \mathsf{Rec}_{\mathcal{M}}(\wp(\mathsf{Val} \times \mathsf{St}) \times \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}))) \\ & \quad \rightharpoonup \wp(\mathsf{Loc} \times \mathsf{St})\end{aligned}$$

$$\begin{aligned}\langle \ell, \sigma\rangle \;\in \mathrm{Spec}_o(A, \{\!|\mathsf{m}_j = \langle B_{\mathsf{m}_j}, T_{\mathsf{m}_j}\rangle|\!\}^{j=1..m}) \;\; & \textit{iff} \;\; \forall \mathsf{m} \in \mathcal{M}.\, \forall v \in \mathsf{Val}.\, \forall \sigma' \in \mathsf{St}. \\ \langle \ell, \sigma\rangle \in A \,\wedge\, ((\langle \ell, \sigma\rangle \in & \mathrm{Spec}_o(A, \{\!|\mathsf{m}_j = \langle B_{\mathsf{m}_j}, T_{\mathsf{m}_j}\rangle|\!\}^{j=1..m}) \\ & \wedge\, \sigma.\ell.\mathsf{m}(\ell, \sigma)) = \langle v, \sigma'\rangle \Rightarrow B_{\mathsf{m}}(v, \sigma') \wedge T_{\mathsf{m}}(\ell', \sigma, v, \sigma'))\end{aligned}$$

The above definition is implicit and due to the contravariant occurrence of Spec_o it is not clear that the specification exists. If all constituent predicates are monotonic one can define the specification to be the fixpoint of the defining functional. But the least fixpoint would not give the intended meaning (as methods would simply be required to be undefined) and the greatest one just leaves us with co-induction as appropriate proof principle which. Co-induction requires an invariant that is often not easy to find or that turns out to be just the specification itself (for an example see [19]).

A more intuitive way to prove objects correct is given in [2]. As the argument of a method do not assume the callee itself but an arbitrary object obeying the object specification.

Definition 9. *The generalised notion of object specification is defined below:*

$$\begin{aligned}\mathrm{Spec}_O : (&(\wp(\mathsf{Loc} \times \mathsf{St}) \rightharpoonup \wp(\mathsf{Loc} \times \mathsf{St})) \times \\ & \mathsf{Rec}_{\mathcal{M}}((\wp(\mathsf{Val} \times \mathsf{St}) \rightharpoonup \wp(\mathsf{Val} \times \mathsf{St})) \times \wp(\mathsf{Loc} \times \mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}}))) \\ & \quad \rightharpoonup \wp(\mathsf{Loc} \times \mathsf{St})\end{aligned}$$

Let $\mathrm{Spec}_o(A, \{\!|\mathsf{m}_j = \langle B_{\mathsf{m}_j}, T_{\mathsf{m}_j}\rangle|\!\}^{j=1..m})$ *be the predicate* $S \in \wp(\mathsf{Loc} \times \mathsf{St})$ *with*

$$
\begin{array}{l}
(\ell,\sigma) \in S \;\equiv\; A(S)(\ell,\sigma) \wedge \\
\quad \forall \mathsf{m} \in \mathcal{M}.\, \forall \ell' \in \mathsf{Loc}.\, \forall \sigma' \in \mathsf{St}.\, \langle \ell', \sigma' \rangle \in S \Rightarrow \\
\quad\quad \forall v \in \mathsf{Val}.\, \forall \sigma'' \in \mathsf{St}. \\
\quad\quad \sigma.\ell.2.\mathsf{m}(\ell', \sigma') = (v, \sigma'') \Rightarrow B_{\mathsf{m}}(S)(v, \sigma'') \;\wedge\; T_{\mathsf{m}}(\ell', \pi_{\mathsf{Val}}(\sigma'), v, \pi_{\mathsf{Val}}(\sigma''))
\end{array}
$$

provided S is unique *with the above property.*

Note that we added some recursive dependencies: A and B_m may use Spec_o recursively. Properties of fields or results that are (locations of) objects themselves can be specified in terms of that object's class type. In the object-based case one cannot refer to a class specification so one must ensure that it is possible to refer to the specification one is about to define recursively.

The following theorem states an existence theorem for object specifications.

Theorem 3. *Given admissible A, B_{m_j} and T_{m_j} as in the definition above where A and B satisfy the following conditions:*

(i) $e : X \subseteq X'$ *implies* $F(e,e) : A(X) \subseteq A(X')$ *for all* $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$.
(ii) *for all* $\mathsf{m} \in \mathcal{M}$, $e : X \subseteq X'$ *implies* $F(e,e) : B_{\mathsf{m}}(X) \subseteq B_{\mathsf{m}}(X')$ *for all* $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$.

where $F(Y,X) = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times Y \rightharpoonup \mathsf{Val} \times X))$ *and* $e : X \subseteq Y$ *abbreviates* $\forall o{:}\mathsf{Loc} \times \mathsf{St}.\ o {\in} X \;\wedge\; e(o){\downarrow} \;\Rightarrow\; e(o) {\in} Y$. *Then for Φ defined as*

$$
\begin{array}{l}
\Phi(Y,X)(\ell,\sigma) \;\equiv\; A(X)(\ell,\sigma) \wedge \\
\quad \forall \mathsf{m} \in \mathcal{M}.\, \forall \ell' \in \mathsf{Loc}.\, \forall \sigma' \in \mathsf{St}.\, \langle \ell', \sigma' \rangle \in Y \Rightarrow \\
\quad\quad \forall v \in \mathsf{Val}.\, \forall \sigma'' \in \mathsf{St}. \\
\quad\quad \sigma.\ell.\mathsf{m}(\ell', \sigma') = (v, \sigma'') \Rightarrow B_{\mathsf{m}}(X)(v, \sigma'') \;\wedge\; T_{\mathsf{m}}(\ell', \pi_{\mathsf{Val}}(\sigma'), v, \pi_{\mathsf{Val}}(\sigma''))
\end{array}
$$

there exists a unique admissible $S \in \wp(\mathsf{Loc} \times \mathsf{St})$ with $S = \Phi(S,S)$.

Proof. The proof uses Andrew Pitts' *relational properties of domains* [14] and can be found in [19].

This theorem guarantees existence only if T_{m} does not refer to the method part of the store. It is yet unknown whether existence can be ensured otherwise (in a less restrictive way). As a consequence, method update is problematic from a specification point of view. The transitions specification cannot express any changes caused by updates if they are not allowed to refer to the method part of the object (store). Method update is considered difficult also in [5] but for a different reason. It is hard to include in the strongly typed encodings.

We are now in a position to explain why we have distinguished between T_{m} and B_{m} in the first place. B_{m} is allowed to refer to the higher-order (method) part of the store whereas T_{m} is not. This has useful consequences as B_{m} can be used to express method invariants. If a specification for $\langle \ell, \sigma \rangle$ uses a B_{m} defined as

$$\langle v, \sigma' \rangle \in B_{\mathsf{m}}(X) \;\equiv\; \langle \ell, \sigma' \rangle \in X$$

it is stipulated that the object ℓ still fulfils the specification (represented by X but later bound by recursion) after execution of m in the new state σ'.

Another advantage is that just by definition of Spec_O the object introduction rule of [2] is correct in our semantics. It explains much better why the calculus is sound than a rather involved induction on derivations.

	class based	object based
object	$\mathsf{Ob} = \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathcal{C}$	$\mathsf{Ob} = \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl})$
store	$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$	$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$
meth's	$\mathsf{Ms} = \mathsf{Rec}_{\mathcal{C}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \times (\mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St}))$	
unfolded	$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathcal{C})$	$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}))$
recursion	fixpoint of $[\![cl]\!] : \mathsf{Ms} \rightharpoonup \mathsf{Ms}$	by self application "through store"
spec type	$\wp(\mathcal{C} \times \mathsf{Ms})$	$\wp(\mathsf{Loc} \times \mathsf{St})$
existence	trivial	only for admissible specs if T_{m} refers to flat part of σ only
proof rule	fixpoint induction	the specification itself
admissibility	required but always guaranteed	required

Table 5. In a nutshell: class-based vs. object-based semantics

5 Comparison

Unfolding the definitions of St for the class-based and the object based language one immediately observes a difference as emphasized in Table 5. It is obvious that the domain of interpretation for the object-based case is itself defined recursively, and that the right hand side of its definition even contains a negative occurrence. This is is a particular instance of a concept known as "recursion through the store". Recursive methods can be implemented without the explicit recursion of the class-based languages by calling some code residing in the store. Specifications are not recursively defined and thus their existence is trivial in the class-based case whereas it needs some heavy domain theoretic machinery to guarantee existence of the object specifications. In the latter case existence requires already the constituent predicates to be admissible and monotonic. Transition specifications must not refer to methods, they may just talk about the fields of objects, but cannot express effects of method updates.

Finally, the main proof rule for the class-based approach is fixpoint induction triggered by the recursive definition of method suites. For the object-calculus the definition of the specifications provides the proof rule that uses as assumption – as "induction hypothesis" – that the specification holds for the arbitrary object that is used as callee of the method in question.

6 Conclusion

The comparison between a typical class-based and a typical object-based language in terms of denotational semantics reveals that the semantic domains and the logic on top of these domains are much more involved in the object-based case. In the latter case, specifications may not exist if method update is allowed. For both languages recursion is involved, although on different levels: for class-based languages only the method suites are recursively defined, for object-based languages the underlying semantic domains and the specifications are recursively defined but not the methods themselves. Due to the recursion, all predicates used for program verification must be admissible in both cases. But only for the class-based language this can be guaranteed in general.

The semantics demonstrate in an intuitive way that crucial proof rules for the logics in [20] and [2] are correct.

Further research includes full proofs of the soundness of the program logics [20, 2] as well as a typed variant of the presented denotational semantics. Other natural extensions include method parameters and access restrictions (data encapsulations). Static methods can easily be incorporated into the class-based approach following the scheme of the nullary constructor. Also abnormal termination (exception handling) and control flow operators could be incorporated (cf. [10]).

One significant advantage of the presented approach is that it does not commit itself to any particular form of logic so other verification calculi can be treated in the very same fashion. In particular it is independent of the underlying (usually first-order) logic. Therefore, our analysis is orthogonal to and thus compatible with the *spatial logic* suggested by [6, 7].

Acknowledgements: Thanks for discussions about class-based and object-based semantics go to Thomas Streicher, but also to Cristiano Calcagno, Peter O'Hearn and Uday Reddy. The anonymous referees gave valuable help to improve the presentation.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
2. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.* Springer, Berlin, 1999.
4. P. America and F.S. de Boer. A proof theory for a sequential version of POOL. Technical Report http://www.cs.uu.nl/people/frankb/Available-papers/spool.dvi, University of Utrecht, 1999.
5. K.B. Bruce, L. Cardelli, and B.C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan*, volume 1281, pages 415–438. Springer LNCS, September 1997.
6. C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Principles and Practice of Declarative Programming*. ACM Press, 2000.
7. C. Calcagno and P.W. O'Hearn. A logic for objects. Talk given in March 2001, 2001.
8. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computations Structures*, volume 1578 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

9. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java. A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 34, pages 132–146. ACM SIGPLAN Notices, 1999.
10. B. Jacobs and E. Poll. A monad for basic java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 150–164. Springer Verlag, 2000.
11. S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.
12. T. Nipkow and D. von Oheimb. Machine-checking the Java Specification: Proving Type-Saftey. In Alves-Foss [3].
13. L.C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.
14. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 321, December 1993.).
15. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods*, 1998.
16. U. Reddy. Objects and classes in Algol-like languages. In *FOOL 5*, 1998. Available at URL http://pauillac.inria.fr/~remy/fool/proceedings.html.
17. B. Reus. A logic of recursive objects. In *Formal Techniques for Java Programs*, volume 251 - 5/1999 of *Informatik Berichte*, pages 58–64. FernUniversität Hagen, 1999.
18. B. Reus. A logic of recursive objects (abstract). In S. Demeyer A. Moreira, editor, *Object-oriented Technology, ECOOP '99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, page 107, Berlin, 1999. Springer.
19. B. Reus and Th. Streicher. Semantics and logics of objects. In *Proceedings of the 17th Symp. Logic in Computer Science*, 2002. To appear. Draft available from `www.in.ormatics.sussex.ac.uk/users/bernhard/papers.html`.
20. B. Reus, M. Wirsing, and R. Hennicker. A Hoare-Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In *FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 300–317, Berlin, 2001. Springer.
21. D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Found. of Software Techn. and Theoret. Comp. Sci.*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.