Modular Semantics and Logics of Classes

Bernhard Reus*

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

Abstract. The semantics of class-based languages can be defined in terms of objects only [8, 7, 1] if classes are viewed as objects with a constructor method. One obtains a store in which method closures are held together with field values. Such a store is also called "higher-order" and does not come for free [13]. It is much harder to prove properties of such stores and as a consequence (soundness of) programming logics can become rather contrived (see [2]).
A simpler semantics separates methods from the object store [4, 12]. But again, there is a drawback. Once the semantics of a package of classes is computed it is impossible to add other classes in a compositional way. Modular reasoning principles are therefore not obtainable either.
In this paper we improve a simple class-based semantics to deal with extensions compositionally and derive modular reasoning principles for a logic of classes. The domain theoretic reasoning principle behind this is fixpoint induction.
Modularity is obtained by endowing the denotations of classes with an additional parameter that accounts for those classes added "later at linkage time."
Local class definitions (inner classes) are possible but for dynamic class-loading one cannot do without higher-order store.

1 Introduction and Motivation

In our quest for a complete denotational understanding of the semantics *and* logics of object-oriented programming languages we have treated object-based languages [13] in terms of Abadi and Cardelli's object calculus [1] and class-based languages [12] in terms of a simple sublanguage of sequential Java.

Class-based languages use the *class* concept to describe a collection of objects that are the *instances* of the class. In [8] two denotational models for various languages (including classes and inheritance) are given. One uses fixpoint closures, the other one self-application but in both models objects contain fields and methods (method closures). This leads to what is called "higher-order store", the domain of which is defined by a mixed variant equation similar to the one below:

$$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}\mathsf{Val}) \ \times \ \mathsf{Rec}_{\mathcal{M}}(\mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}) \ .$$

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

For such stores invariance properties are hard to show.

A simpler approach aims at separating methods from objects. Methods do *not* have to be stored in the heap but are gathered in an environment that contains a method suite for each class. The class name acts as a link between the objects in the store and the method suites and is used for method dispatch. There are several variations of such a denotational semantics in the literature, [3, 4, 12], but in all of them only a *fixed* set of classes can be interpreted. Logics based on *operational semantics* which address these problems were presented in [11, 14]. A *denotational semantics*, however, provides easier handling of mutual recursive modules as well as better tools for analysing and extending the language and logic. It also allows us to study the differences between logics for languages with higher-order store (object-based languages) and class-based languages more thoroughly.

In this paper we introduce a *denotational* semantics that is "open" with respect to extensions of classes and thus allows for a modular programming logic. The main contributions are:

- modular (compositional) denotational semantics for classes without higher-order store
- modular logic for classes (in terms of denotations)
- semantics for local classes
- explanation of modularity in class-based languages and relationship w.r.t. object-based languages.

Correctness of programs is usually shown using Hoare-like calculi. But it is the design, analysis, and soundness proof of these calculi that can be simplified using denotational semantics.

In the next section we briefly present the syntax of a simple class-based language. Then a (standard) denotational semantics is provided. Section 4 improves this semantics to obtain modularity and compositionality. Finally, modular proof rules are discussed. The paper end with a short summary and outlook (Section 6).

2 Syntax

The syntax and semantics of the class-based language is a refinement of the one in [12]. Let $\mathcal{F}$ be the set of field names, $\mathcal{M}$ the set of method names, and $\mathcal{C}$ the set of class names. Usually, f stands for a field name, m, n stand for method names, and C, D for class names.

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

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 always have to return a result.

Additionally, there is a syntax for class definitions:

$$\begin{array}{lll} c & ::= \texttt{class}\ \mathsf{C}\ \{[\texttt{inherits}\ \mathsf{D}]\ \mathsf{f}_i^{\ i=1,\ldots,n} \\ & & \mathsf{m}_j = b_j^{\ j=1,\ldots,m} \\ & & \mathsf{C}\,() = b \\ & & \} \\ cl & ::= \epsilon \mid c\ cl \end{array}$$

A class definition is similar to one in Java with all type declarations omitted. It contains a list of field declarations, a list of method declarations and a constructor. For the sake of simplicity all functions and the constructor are nullary and shadowing of field variables by inheritance is disallowed, ie. it is not possible to declare a field in a subclass that was already declared in a superclass[1].

3 Denotational Semantics

The denotational semantics is supposed to live in the category of predomains – ie. complete partially ordered sets – and continuous maps. Equivalently one can work in a category with domains (that have a least element) and strict, continuous maps. Note that the fixpoint operator fix is only defined on domains with least element.

Indexed collections, like stores and method suites, can be nicely modelled as records.

3.1 Record types

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 is easily seen that $\mathsf{Rec}_{\mathbb{L}}$ is a locally continuous functor on PreDom. A record with labels l_i and corresponding entries v_i $(1 \leq i \leq k)$ is written $\{\!|l_1 = v_1, \ldots l_k = v_k|\!\}$. Notice that $\mathsf{Rec}_{\mathbb{L}}(A)$ is always non–empty as it contains the element $\langle \emptyset, \emptyset \rangle$.

Records are ordered as follows:

$$\langle L_1, f_1 \rangle \sqsubseteq \langle L_2, f_2 \rangle \Leftrightarrow L_1 = L_2 \ \wedge\ \forall \ell \in L_1.\, f_1(\ell) \sqsubseteq_A f_2(\ell)$$

[1] A more sophisticated treatment shadowing the duplicate fields of the superclass as used in Java can be modelled by adding the duplicate fields with appropriately changed field names.

Therefore, a record and its extension are *in*comparable. Note that records do not have a least element, only minimal ones. This can be remedied by lifting the record type to attach a new least element, usually called $\bot$: $\mathsf{Rec}_{\mathbb{L}}(A)$ is a complete poset if A is so, thus the lifted $\mathsf{Rec}_{\mathbb{L}}(A)_{\bot}$ is always a domain.

Basic record operations like selection, update, extension, and deletion are defined below.

Definition 1. *Let $r \in \mathsf{Rec}_{\mathbb{L}}(A)$ such that $r = \langle L, f\rangle$ with $L \subseteq \mathbb{L}$ and $f \in A^L$.*
Definedness of label l in record r is as follows: $l \in \mathsf{dom}\, r \Leftrightarrow l \in L$.
Selection of a label $l \in \mathbb{L}$ in record r, short $r.l$, is defined if $l \in \mathsf{dom}\, r$ and yields $f(l) \in A$.

An update function for records is define in Table 1. It is undefined for labels which do not appear in the argument record.

$$\{\!|l_i{=}f_i|\!\}^{i=1\ldots n}[l{:=}f] = \begin{cases} \text{undefined} & \text{if } l \notin \mathsf{dom}\{\!|l_i{=}f_i|\!\}^{i=1\ldots n} \\ \{\!|l_1{=}f_1, \ldots, l_i{=}f, \ldots, l_n{=}f_n|\!\} & \text{if } l = l_i \end{cases}$$

Table 1. Definition of record update

If $r_1 = \langle L_1, f_1\rangle$ and $r_2 = \langle L_2, f_2\rangle$ are two records in $\mathsf{Rec}_{\mathbb{L}}(A)$ then the extension $r_1 + r_2$ is defined as follows:

$$\langle L_1, f_1\rangle + \langle L_2, f_2\rangle = \langle L_1 \cup L_2, \lambda l.\, \textit{if } l \in L_1 \textit{ then } f_1(l) \textit{ else } f_2(l)\rangle$$

Note that in $r_1 + r_2$ the values of the fields in r_2 which are also used in r_1 are lost.

Finally, if $r = \langle L, f\rangle$ and $l \in \mathbb{L}$ then the deletion operation which erases l in r, $r \setminus l$, is defined as follows:

$$r \setminus l = \begin{cases} r & \textit{if } l \notin \mathsf{dom}\, r \\ \langle L \setminus l, f\rangle & \textit{otherwise} \end{cases}$$

For a nested record $r \in \mathsf{Rec}_{\mathbb{L}_1}(\mathsf{Rec}_{\mathbb{L}_2}(A))$ we abbreviate $r[l_1 := r.l_1[l_2 := a]]$ by the simpler and more intuitive $r[l_1.l_2 := a]$.

3.2 Semantic Domains

We assume flat predomains for basic values (like booleans or integers) Val and a flat predomain of locations Loc.

The class-based language introduced above finds its interpretation within the following *non-recursive* system of domains:

Definition 2. *Let the semantic counterparts of objects (1), stores (2), closures (3), method suites (4), and class descriptors (5) be defined as below.*

$$\mathsf{Ob} = \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathcal{C} \tag{1}$$

$$\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}) \tag{2}$$

$$\mathsf{Cl} = \mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St} \tag{3}$$

$$\mathsf{Ms} = \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \times (\mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{St}) \times \mathcal{C} \tag{4}$$

$$\mathsf{Mss} = \mathsf{Rec}_{\mathcal{C}}(\mathsf{Ms}) \tag{5}$$

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 *store* is a record of objects indexed by locations. 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. A *method suite* (Ms) is a record $\mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \times (\mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{St}) \times \mathcal{C}$ which contains all methods of a class, the initialisation code for the constructor, and the name of the superclass. We assume that there is always a superclass, ie. there is a root class `Root` that all classes inherit from[2]. The constructor takes a location in which to create the object. This reference is created freshly by the semantics of `new` and then passed to the constructors of the superclass. A *class descriptor* (Mss) is the collection of method suites for all declared classes. The function $\mathsf{type} : \mathsf{Ob} \to \mathcal{C}$ returns an object's class, ie. $\mathsf{type}(o) = o.2$, which is the object's dynamic type. For interpreting variables we need an environment Env which maps variables to values, ie. $\mathsf{Env} = \mathsf{Var} \rightharpoonup \mathsf{Val}$.

3.3 Semantic Equations

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 $\mu \in \mathsf{Mss}$ and an expression a, its interpretation $[\![a]\!]\rho : \mathsf{Mss} \to \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}$ 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.*
Given an environment of method suites $\mu \in \mathsf{Mss}$ a class definition c is defined via $[\![c]\!] : \mathsf{Mss} \to \mathsf{Ms}$ in Table 3 (page 6).

The methods of the superclass are copied into the method suite of the subclass. It is more memory efficient not to copy them and use a more sophisticated method dispatch instead that searches in super classes. Finally, since we work in an untyped setting, it is legitimate to initialize any field with zero. In a typed setting zero would be replaced by a default value for the type of the field. It is no contradiction to the untypedness of our semantics to have classes (class names) stored in objects as these classes represent *runtime* types needed for dynamic dispatch. There are no static types in use.

[2] In Java this would be `Object`.

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

Table 2. Denotational semantics of expressions

$$
\left[\!\!\left[\begin{array}{l}\texttt{class C inherits D } \{ \\ \quad \mathsf{f}_i{}^{\,i=1..n} \\ \quad \mathsf{m}_j{=}b_j{}^{\,j=1..m} \\ \quad \texttt{C() =}b \\ \quad \}\end{array}\right]\!\!\right] \mu = \left\langle \begin{array}{l} \{\!|\mathsf{m}_j{=}\lambda\langle \ell,\sigma\rangle.\,[\![b_j]\!]\,(\text{this}{\mapsto}\ell)\,\mu\,\sigma|\!\}^{j=1..m} + \mu.\mathsf{D}.1, \\ \lambda\langle \ell,\sigma\rangle.\,([\![b]\!]\,(\text{this}{\mapsto}\ell)\,\mu\,(\mu.\mathsf{D}.2(\ell,\sigma))).2[\mathsf{f}_i := 0], \\ \mathsf{D}\end{array}\right\rangle
$$

Table 3. Denotational semantics of class definitions

The semantics of a *module* `class C` ... could be defined as

$$\mathsf{fix}\ \mu : \mathsf{Mss}_\perp .\ \{\!|\mathsf{C} = [\![\texttt{class C}\ldots]\!]\,\mu|\!\} \quad .$$

Fixpoint induction would then provide the corresponding proof principle. But this definition does not give the desired result if classes depend on other classes. An example is used below to emphasize the problem:

```
class Top = {
 f
 m() = this.f.n()
 n() = 0
 Top() = this.f = this
}

class C inherits Top {
 n() = 1
 C() = this = this  /* skip */
}
```

The semantics of these two classes according to the above are:

$$
\begin{array}{rl}
[\![\texttt{class Top}\ldots]\!] = \mathsf{fix}\ \mu.\ \{\!|\mathsf{Top} = \langle\{\!| & \mathsf{m} = \lambda\langle \ell,\sigma\rangle.\mu.\mathsf{type}(\sigma.(\sigma.\ell.\mathsf{f})).1.\mathsf{n}(\sigma.\ell.\mathsf{f},\sigma), \\
 & \mathsf{n} = \lambda\langle \ell,\sigma\rangle.\,0|\!\}, \\
 & \lambda\langle \ell,\sigma\rangle.\,\sigma[\ell := \langle\{\!|\mathsf{f} = \ell|\!\}, \mathsf{Top}\rangle], \mathsf{Root}\rangle\ |\!\}
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{class C inherits Top}\ldots]\!] = \\
\quad \mathsf{fix}\ \mu.\ \{|\mathtt{C} = \langle \quad \{|\mathsf{m} = \lambda\langle\ell,\sigma\rangle.\ \mu.\mathsf{type}(\sigma.(\sigma.\ell.\mathsf{f})).1.\mathsf{n}(\sigma.\ell.\mathsf{f},\sigma), \\
\qquad\qquad\qquad\quad \mathsf{n} = \lambda\langle\ell,\sigma\rangle.\ 1|\}, \\
\qquad\qquad\qquad\quad \lambda\langle\ell,\sigma\rangle.\ \sigma[\ell := \langle\{|\mathsf{f} = \ell|\}, \mathtt{C}\rangle], \mathsf{Top}\ \rangle\ |\}
\end{array}
$$

If we join the two classes into one class descriptor μ defined as follows

$$\mu = \{|\mathtt{Top} = [\![\texttt{class Top}\ldots]\!]\} + \{|\mathtt{C} = [\![\texttt{class C inherits Top}\ldots]\!]\}$$

and interpret the following expression t

```
let o = new Top()
in let _ = o.f = new C()
   in  o.m()
```

w.r.t. μ we obtain:

$$
\begin{array}{rl}
[\![t]\!]\,\emptyset\,\mu\,\sigma = & \textbf{let}\ \sigma_1 = \sigma[\ell := \langle\{|\mathsf{f} = \ell|\}, \mathtt{Top}\rangle]\ \textbf{in} \\
& \textbf{let}\ \sigma_2 = \mu.\mathtt{Top}.2(\ell', \sigma_1)[\ell'.2 := \mathtt{C}]\ \textbf{in} \\
& \textbf{let}\ \sigma_3 = \sigma_2[\ell.\mathsf{f} := \ell']\ \textbf{in} \\
& \mu.\mathsf{type}(\sigma_3.\ell).1.\mathsf{m}(\ell, \sigma_3)
\end{array}
$$

where ℓ is fresh in σ and ℓ' is fresh in σ_1. This simplifies to

$$\textbf{let}\ \sigma_4 = \sigma[\ell := \langle\{|\mathsf{f} = \ell'|\}, \mathtt{Top}\rangle, \ell' := \langle\{|\mathsf{f} := \ell'|\}, \mathtt{C}\rangle]\ \textbf{in}\ \mu.\mathtt{Top}.1.\mathsf{m}(\ell, \sigma_4)$$

which by the semantics of class `Top` simplifies to

$$\mu.\mathsf{type}(\sigma_4.(\sigma_4.\ell.\mathsf{f})).1.\mathsf{n}(\sigma_4.\ell.\mathsf{f}, \sigma_4)$$

which in turn evaluates to

$$\mu.\mathtt{C}.1.\mathsf{n}(\ell', \sigma_4) \qquad .$$

Obviously, the result depends on the semantics of the method `n` in `C`, ie. $\mu.\mathtt{C}.1.\mathsf{n}$. But the fixpoint operation yields no defined result for $\mu.\mathtt{C}$, just one for $\mu.\mathtt{Top}$ and thus the result is undefined whereas it should be 1. The reason for this unexpected behaviour is that the fixpoint has closed the method suites available. It cannot be updated to contain the methods of other classes added later.

To avoid this problem we define the semantics of classes slightly differently, in a parameterized way.

4 A modular semantics for classes

Modules depend on a context of class declarations unknown at the time of definition. Later they may be linked together with this context changing the semantics of the original modules. This effect will be modelled by (mutual) fixpoints. Recall that records defined by fixpoints can not get bigger (because of their invariance under extensions) but the values of their fields can become more defined. This is enough for our purposes, since in module or package definitions the number of methods and fields is fixed. Nevertheless, the record extension operator is needed to express a temporarily growing number of classes.

Definition 4. *We define a polymorphic operator maprec that applies a function to all components of a record: $maprec_A$:* $(A \to B) \to \mathsf{Rec}_{\mathbb{L}}(A) \to \mathsf{Rec}_{\mathbb{L}}(B)$ *as follows:*

$$\mathrm{maprec}\, f\, \langle D, g\rangle = \langle D, f \circ g\rangle$$

which is in analogy with map for lists.

A modular class definition has the following semantics taking into account further packaging with other classes:

Definition 5. *The semantics* $[\![\_]\!]^{\mathrm{m}} : \mathsf{Mss} \to \mathsf{Ms}$ *is defined as*

$$[\![\mathtt{class}\ \mathsf{C}\ \ldots]\!]^{\mathrm{m}} = \lambda\mu : \mathsf{Mss}.\, \mathsf{fix}\,\tau : \mathsf{Ms}_{\perp}.\, [\![\mathtt{class}\ \mathsf{C}\ \ldots]\!]\ (\{\!|\mathsf{C} = \tau|\!\} + \mu)$$

where the parameter μ represents the possible additional classes C *may depend on.*

Even if C is not intended to refer to any other class, in an open world it must reserve the right to refer to future *subclasses* of itself.

A "package" or "linkage" operator is needed that links a module with a given package, ie. a list of linked (or packed) modules. Like modules packages have an additional parameter in order to be open to future extensions. In order to accomplish this, we define the type

$$\mathsf{CTs} = \mathsf{Rec}_{\mathcal{C}}(\mathsf{Mss} \to \mathsf{Ms})$$

which represents collections of class transformers of type $\mathsf{Mss} \to \mathsf{Ms}$ that describe the semantics of a class depending on a class descriptor for the context.

Definition 6. *The package operator*

$$\mathrm{pack}_{\mathsf{C}} : (\mathsf{Mss} \to \mathsf{Ms}) \times \mathsf{CTs} \to \mathsf{CTs}_{\perp}$$

is defined as follows:

$$\begin{aligned}\mathrm{pack}_{\mathsf{C}}(M, P) = \mathsf{fix}\ \delta : \mathsf{CTs}_{\perp}.\ & \\ & \{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\, M(\delta(\mu) \setminus \mathsf{C} + \mu)|\!\} + \\ & \mathrm{maprec}\,(\lambda x : \mathsf{Mss} \to \mathsf{Ms}.\, \lambda\mu : \mathsf{Mss}.\, x(\delta(\mu).\mathsf{C} + \mu))\, P\ .\end{aligned}$$

Definition 7. *The semantics of a package of class definitions can be defined as follows:*

$$[\![\mathtt{class}\ \mathsf{C}\{\ldots\}\ cl]\!] = \mathrm{pack}_{\mathsf{C}}([\![\mathtt{class}\ \mathsf{C}\{\ldots\}]\!]^{\mathrm{m}}, [\![cl]\!])$$

and $[\![\epsilon]\!] = \{\!|\!|\!\}$.

Lemma 1. *Packaging with nothing does not have any effect, ie.*

$$\mathrm{pack}([\![\mathtt{class}\ \mathsf{C} = \ldots]\!]^{\mathrm{m}}, \emptyset) = \{\!|\mathsf{C} = [\![\mathtt{class}\ \mathsf{C} = \ldots]\!]^{\mathrm{m}}|\!\}$$

Proof. Let $M = [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}$ and assume $\delta = \{\!|\mathsf{C} = M|\!\}$ (1). If we can show that

$$\{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\ [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}(\delta(\mu) \setminus \mathsf{C} + \mu)|\!\} = \{\!|\mathsf{C} = M|\!\}$$

then by fixpoint induction we have shown

$$\mathsf{fix}\,\delta : \mathsf{CTs}_\perp.\ \{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\ [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}(\delta(\mu) \setminus \mathsf{C} + \mu)|\!\} = \{\!|\mathsf{C} = M|\!\}$$

which implies the claim.

$$\begin{array}{l}\{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\ [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}(\delta(\mu) \setminus \mathsf{C} + \mu)|\!\} =_{(1)} \\ = \{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\ [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}(\{\!|\mathsf{C} = M(\mu)|\!\} \setminus \mathsf{C} + \mu)|\!\} = \\ = \{\!|\mathsf{C} = \lambda\mu : \mathsf{Mss}.\ [\![\texttt{class C} = \ldots]\!]^{\mathrm{m}}\ \mu|\!\} = \\ = \{\!|\mathsf{C} = M|\!\} \quad .\end{array}$$

One can always "close" a package or module by just applying it to the empty environment.

4.1 Inner Classes

Inner classes are classes defined "on-the-fly" inside classes or methods. Simple class descriptors are not sufficient as argument to the interpretation function since packaging needs transformers. The type of the interpretation function thus changes to

$$[\![\!]\!]\rho : \mathsf{CTs} \to \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St} \quad .$$

If "$\texttt{let class C} \ \ldots \texttt{in}\ s$" denotes the syntax for local inner class declarations the interpretation is as follows:

$$[\![\texttt{let class C}\ \ldots\ \texttt{in}\ s]\!]\ \rho\,\delta\,\sigma = [\![s]\!]\ \rho\ \mathrm{pack}_{\mathsf{C}}([\![\texttt{class C}\ \ldots]\!]^{\mathrm{m}}, \delta)\,\sigma$$

Whenever the interpretation really needs the method environment – ie. when evaluating a method call or creating an object – the environment can be closed for this moment of time to find the right closure:

$$\begin{array}{lll}[\![\texttt{new C()}]\!]\,\rho\,\delta\,\sigma & = \langle \ell, (\delta.\mathsf{C}\ \{\!|\!|\!\}).2(\ell, \sigma)[\ell.2 := \mathsf{C}]\rangle & \\ & \quad \text{where } \ell \notin \mathsf{dom}\,\sigma & \\ [\![a.\mathsf{m}()]\!]\,\rho\,\delta\,\sigma & = \textbf{let } \langle \ell, \sigma'\rangle = [\![a]\!]\,\rho\,\delta\,\sigma \textbf{ in } (\delta.\mathsf{type}(\sigma'.\ell)\ \{\!|\!|\!\}).1.\mathsf{m}(\ell, \sigma') & \end{array}$$

The corresponding change to method store transformers means that we have to change the semantics of the class modules slightly:

$$\begin{array}{l}[\![\texttt{class}\,C\ \ldots]\!]^{\mathrm{m}} = \lambda\mu : \mathsf{Mss}.\,\mathsf{fix}\,\tau : \mathsf{Ms}_\perp.\,[\![\texttt{class}\,C\ \ldots]\!] \\ \qquad\qquad \{\!|C = \lambda\_ : \mathsf{Mss}.\,\tau|\!\} + \mathrm{maprec}\,(\lambda x : \mathsf{Ms}.\,\lambda\_ : \mathsf{Mss}.\,x)\,\mu\end{array}$$

5 Logics of Programs

Having fixed the denotational semantics of classes we can start reasoning about denotations. This is in analogy with the LCF (Logic of Computable Functions) project (see e.g. [10]) for the functional paradigm.

5.1 Specifications

First we have to define an appropriate notion of *specification* for classes following [6, 11, 14]. Compare also with [12].

As every class contains 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.

$$\begin{array}{lll} B_{\mathsf{m}} \in \wp(\mathsf{Val} \times \mathsf{St}) & \text{result specification} & \text{for } \mathsf{m} \in \mathcal{M} \\ T_{\mathsf{m}} \in \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}) & \text{transition specification} & \text{for } \mathsf{m} \in \mathcal{M} \end{array}$$

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."

A modular specification may depend on the specification of some other classes. The existence of other classes in the environment may indeed be part of the specification itself.

Definition 8. *Specification building operators for methods (*mth*), classes (*cls*), and packages (*pck*) are defined below. For the sake of readability we abbreviate the type of result specifications* $\mathsf{RSpec} = \wp(\mathsf{Val} \times \mathsf{St})$ *and transition specifications* $\mathsf{TSpec} = \wp(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St})$.

$$\begin{array}{l} \mathrm{mth} : \mathsf{RSpec} \times \mathsf{TSpec} \to \ \wp(\mathcal{C} \times \mathsf{Cl}) \\ \mathrm{cls} \ \ : \mathsf{Rec}_{\mathcal{M}}(\mathsf{RSpec} \times \mathsf{TSpec}) \ \to \ \wp(\mathcal{C} \times \mathsf{Ms}) \\ \mathrm{pck} \ : \mathsf{Rec}_{\mathcal{C}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{RSpec} \times \mathsf{TSpec})) \to \ \wp(\mathsf{Mss}) \end{array}$$

$$\begin{array}{l} \langle \mathsf{C}, f \rangle \in \mathrm{mth}(B,T) \ \textit{iff} \ \forall \ell \in \mathsf{Loc}.\, \forall v \in \mathsf{Val}.\, \forall \sigma, \sigma' \in \mathsf{St}. \\ \qquad\qquad (\mathsf{type}(\sigma.\ell) = \mathsf{C} \ \wedge \ f(\ell,\sigma) = \langle v, \sigma' \rangle) \\ \qquad\qquad \Rightarrow B(v,\sigma') \wedge T(\ell,\sigma,v,\sigma') \end{array}$$

$$\langle \mathsf{C}, \mu \rangle \in \ \mathrm{cls}(R) \quad \textit{iff} \ \forall \mathsf{m} \in \mathcal{M}.\ \mathsf{m} \in \mathsf{dom}\, R \Rightarrow \mathsf{m} \in \mathsf{dom}\, \mu \ \wedge \ \langle \mathsf{C}, \mu.\mathsf{m} \rangle \in \mathrm{mth}(R.\mathsf{m})$$

$$\delta \ \in \ \mathrm{pck}(\boldsymbol{R}) \quad \textit{iff} \ \forall \mathsf{C} \in \mathcal{C}.\ \mathsf{C} \in \mathsf{dom}\, \boldsymbol{R} \Rightarrow \mathsf{C} \in \mathsf{dom}\, \delta \ \wedge \ \langle \mathsf{C}, \delta.\mathsf{C} \rangle \in \mathrm{cls}(\boldsymbol{R}.\mathsf{C})$$

The following abbreviation will be used repeatedly:

$$\gamma \in \mathsf{C} \mapsto R \text{ iff } \mathsf{C} \in \mathsf{dom}\, \gamma \ \wedge \ \langle \mathsf{C}, \gamma.\mathsf{C} \rangle \in \mathrm{cls}(R)$$

Proposition 1. *The specifications above are admissible predicates.*

Proof. For example,

$$\begin{array}{l} \langle \mathsf{C}, f \rangle \in \mathrm{mth}(B,T) \ \text{ iff } \ \forall \ell \in \mathsf{Loc}.\, \forall v \in \mathsf{Val}.\, \forall \sigma, \sigma' \in \mathsf{St}. \\ \qquad\qquad (\mathsf{type}(\sigma.\ell) \neq \mathsf{C} \ \vee \ f(\ell,\sigma)\!\uparrow \vee (B(f(\ell,\sigma)) \wedge T(\ell,\sigma,f(\ell,\sigma)))) \end{array}$$

As $\mathcal{C}$ 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, and T are admissible. The former is by definition, the latter are because they are predicates on a flat predomain. One can show similarly that the other predicates are admissible. Note that $\mathsf{C} \in \mathsf{dom}\,\gamma$ is admissible in γ due to the non-extension order on records.

Specifications for class description transformers can also be defined in the logical relations style:

Definition 9. *Given predicates $P \in \wp(A)$ and $Q \in \wp(B)$, a predicate $P \to Q \in \wp(A \to B)$ is defined as follows:*

$$f \in P \to Q \text{ iff } \forall a \in A.\ a \in P \Rightarrow f(a) \in Q\ .$$

Corollary 1. *If Q is admissible so is $P \to Q$.*

5.2 Modular Proof Rules

As the semantics of classes and packages is defined via fixpoints it seems adequate to use fixpoint induction. Fortuitously, as shown above, the specifications in use are admissible such that fixpoint induction is applicable.

A spatial conjunction operator, $*$, for records (of method suites) will be used below. It is defined in analogy with separation logic [9, 5] where the operator works on heaps. It is more appropriate than normal conjunction $\wedge$ due to the modularity of class definitions.

Definition 10. *Let $P, Q \in \wp(\mathsf{Mss})$, then $P * Q \in \wp(\mathsf{Mss})$ is defined as follows:*

$$\gamma \in P * Q \text{ iff } \exists \gamma_1, \gamma_2 : \mathsf{Mss}.\ \mathsf{dom}\,\gamma_1 \cap \mathsf{dom}\,\gamma_2 = \emptyset \wedge \gamma = \gamma_1 + \gamma_2 \wedge \gamma_1 \in P \wedge \gamma_2 \in Q\ .$$

We assume that $*$ has higher precedence than $\to$.

Admissibility of $P * Q$ is a more delicate matter but fortunately it is not required for the rules below since $*$ always appears on the left hand side of an implication $\to$.

The rules below show which assumptions can be used when in order to prove packages which are broken down into classes. The correctness of classes under certain context assumptions can the be proved using standard techniques not discussed in this paper.

Theorem 1. *The following proof rules are correct:*

$$(1)\ \frac{[\![\texttt{class C } \ldots]\!] \in (\mathsf{C} \mapsto R) * \Gamma \to \mathrm{cls}(R)}{[\![\texttt{class C } \ldots]\!]^{\mathrm{m}} \in \Gamma \to \mathrm{cls}(R)}$$

$$(2)\ \frac{M \in \Gamma * \mathrm{pck}(\boldsymbol{R'}) \to \mathrm{cls}(R) \qquad P \in (\mathsf{C} \mapsto R) * \Gamma\ \to \mathrm{pck}(\boldsymbol{R'})}{\mathrm{pack}_{\mathsf{C}}(M, P) \in \Gamma \to \mathrm{pck}(\{\!|\mathsf{C} = R|\!\} + \boldsymbol{R'})}$$

$$(3)\ \frac{[\![c]\!] \in (\mathsf{C} \mapsto R) * \mathrm{pck}(\boldsymbol{R'}) * \Gamma \to \mathrm{cls}(R) \qquad [\![cl]\!] \in (\mathsf{C} \mapsto R) * \Gamma \ \to \mathrm{pck}(\boldsymbol{R'})}{[\![c\ cl]\!] \in \Gamma \to \mathrm{pck}(\{\!|\mathsf{C} = R|\!\} + \boldsymbol{R'})}$$

Proof. Since the predicates in use are admissible one obtains correctness by fixpoint induction.
For rule (1) assume (i) that $\mu \in \Gamma$ and (induction hyp) that $\tau \in \mathrm{pck}(R)$. We have to show that $[\![\texttt{class C}\ldots]\!](\{\!|\mathsf{C} = \tau|\!\} + \mu) \in \mathrm{pck}(R)$. But this holds by the premise of rule (1) since $\{\!|\mathsf{C} = \tau|\!\} + \mu \in C \mapsto R * \Gamma$ by (i) and (ind.hyp.).
Rules (2) and (3) can be shown similarly with (3) using (2).

The following predicate states that a certain class in a class descriptor is a subclass of another. The predicate is well-defined since the subtype hierarchy is not circular.

Definition 11. *Let* $\mathsf{C}, \mathsf{D} \in \mathcal{C}$ *and* $\gamma \in \mathsf{Mss}$*:*

$$\mathsf{D} \leq_\gamma \mathsf{C} \ \textit{iff}\ \gamma.\mathsf{D}.3 = \mathsf{C} \vee (\gamma.\mathsf{D}.3 = \mathsf{D'} \wedge \mathsf{D'} \leq_\gamma \mathsf{C})\ .$$

5.3 Method Invocation and Inheritance

The semantics of method invocation depends on the (dynamic) class type of the callee object. The class is only known, however, for `this`, it is not known for arbitrary objects on the heap. To verify a property of the method it could be necessary to stipulate that the method behaves identically in all classes. This is unrealistic. It is more convenient to assume that it behaves identically for a certain branch of the class hierarchy. If it is known that the maximal class type of an object is C, then the possible methods to be taken into account can be reduced to those in the hierarchy below C. For all of those one could stipulate that they share some behaviour sufficient to prove the given specification. This can be done using the spatial implication operator $\triangleright$ (see also [9]):

Definition 12. *For any* $\gamma \in \mathsf{Mss}$ *and* $P, Q \in \wp(\mathsf{Mss})$ *we have*

$$\gamma \in P \triangleright Q \quad \textit{iff}\ \ \forall \beta : \mathsf{Mss}.\ \beta \in P \Rightarrow \gamma + \beta \in Q\ .$$

With the help of $\triangleright$ one can now express that no extension of a context may contain a subclass of C that does not fulfill ϕ, briefly $\mathit{Sub}(C, \phi) \triangleright \mathit{false}$ where $\mathit{Sub}(C, \phi)$ is defined as shown below:

$$\gamma \in \mathit{Sub}(\mathsf{C}, \phi) \text{ iff } \exists \mathsf{D} : \mathcal{C}.\ \mathsf{D} \in \mathsf{dom}\,\gamma\ \wedge\ \gamma.\mathsf{D} \notin \phi\ \wedge \mathsf{D} <_\gamma \mathsf{C}\ .$$

Thus, for a proof of $\delta \in \Gamma * \mathit{Sub}(C, \phi) \triangleright \mathit{false} \to \Delta$ one can assume that the input $\mu \in \Gamma * \mathit{Sub}(C, \phi) \triangleright \mathit{false}$ does not contain any subclasses of C that do not fulfill ϕ. In order to employ behavioural subtyping for C one chooses ϕ to be the specification of C.

6 Conclusion

We have presented a modular denotational semantics for a class-based language and sound proof rules for managing the modules. No recursive domains are necessary as long as persistent classes are not loaded at run time. To achieve the latter one could resort to the approach presented in [13] for object-based languages, paying however the price of a much more involved semantics and logics with more severe restrictions.

The advantage of the semantics presented in this paper is that fixpoint induction is sufficient to derive useful proof rules for modular specifications, the only restriction being that specifications are admissible. Furthermore, the denotational approach does not commit itself to a particular logic so the techniques are applicable to all kinds of languages.

Future work comprises the integration of data abstraction such that accessibility restrictions can be modelled. The applicability of the presented technique to separation logic needs to be investigated as much as the benefit of using separation logic for the records of class definitions.

Acknowledgements: The idea of this paper emerged in discussion with Peter O'Hearn and Uday Reddy about the alleged limitations of a simple denotational semantics for classes. Thanks to Thomas Streicher and Hubert Baumeister for useful remarks and pointers.

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. 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.
4. Anindya Banerjee and David Naumann. Representation independence, confinement, and access control. In *Proceedings of ACM Principles of Programming Languages POPL*, volume 164, pages 166–177. ACM press, 2002.
5. Cristiano Calcagno and Peter W. O'Hearn. On garbage and program logic. In *FoSSaCS*, volume 2030 of *LNCS*, pages 137–151, Berlin, 2001. Springer.
6. 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.
7. Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In *Proceedings Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, 1991.
8. 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.

9. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, Berlin, 2001. Springer.
10. L.C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.
11. 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.
12. B. Reus. Class based vs. object based: A denotational comparison. In *Algebraic Methodology And Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 473–488, Berlin, 2002. Springer Verlag.
13. B. Reus and Th. Streicher. Semantics and logics of objects. In *Proceedings of the 17th Symp. Logic in Computer Science*, pages 113–122, 2002.
14. 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.