Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 316 (2004) 191–213

Theoretical Computer Science

www.elsevier.com/locate/tcs

Semantics and logic of object calculi

Bernhard Reus[a,*,1], Thomas Streicher[b]

[a]Department of Informatics, University of Sussex, Falmer, Brighton, UK

[b]Fachbereich Mathematik, TU Darmstadt, Germany

Abstract

The main contribution of this paper is a formal characterization of recursive object specifications and their existence based on a denotational untyped semantics of the object calculus. Existence is not guaranteed but can be shown employing Pitts' results on relational properties of domains. The semantics can be used to analyse and verify Abadi and Leino's object logic but it also suggests extensions. For example, specifications of methods may not only refer to fields but also to methods of objects in the store. This can be achieved without compromising the existence theorem. An *informal* logic of predomains is in use intentionally in order to avoid any commitment to a particular syntax of specification logic.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Object logic; Programming logic; Program verification; Denotational semantics; Domain theory

1. Introduction and motivation

Programming logics have been suggested for object calculi [2,12] as well as object-oriented class-based programming languages [6,8,11,15,16,21]. Since objects are inherently recursive due to self-reference in method calls, their specifications are recursive, too. Soundness proofs for programming logics for object calculi w.r.t. operational semantics can thus become rather involved. Moreover, existence of object specifications is generally neglected although a subtle point. In general, the meaning of a specification is fully described by its introduction rule for object formation. Therefore, the existence of the specification is equivalent to the validity of its introduction rule. The resulting

* Corresponding author.
E-mail addresses: bernhard@sussex.ac.uk (B. Reus), streicher@mathematik.tu-darmstadt.de (T. Streicher).
[1] Partially supported by the EPSRC under Grant GR/R65190/01 and the Nuffield Foundation under Grant NAL/00244/A.

implicit definition of a specification $S = \Phi(S)$ neither guarantees existence nor uniqueness unless Φ is of a certain form.[2] Domain theory provides sufficient machinery to guarantee existence *and* uniqueness. Therefore, working with a denotational semantics puts us into a position to precisely account for this problem.

As far as the authors are aware, such a systematic denotational analysis of object logics has not been carried through yet although there is a successful role model, the *logic of computable functions* (LCF) logic, for the functional paradigm.

The outline of this paper is as follows. First, a denotational semantics of the functional and imperative untyped object calculus of Abadi and Cardelli [1] is given in Section 2. Having done this, a notion of specification inspired by the Abadi and Leino logic [2] can be defined on the resulting object domains (Section 3). In Section 4, we prove existence of these specifications under mild assumptions employing Pitts' machinery for relational properties of domains [14]. One of these assumptions can even be dropped if the method specifications follow certain patterns (Section 4.2.2). An example of a recursive specification definition that does not have a fixpoint is given in Section 4.3.

The existence theorem is not only interesting in its own right, it can also be applied to

- prove soundness of the object formation rule in [2] (in an untyped way but types can be encoded as specifications);
- exemplify via counterexamples that certain recursive specifications cannot exist (or more precisely that certain recursive definitions do not have a fixpoint);
- suggest extensions of an existing programming logic [2,12] introducing method invariants and method update;
- suggest treatment of general higher order store (involving code pointers).

The first two items are discussed in Section 5. The proposed technique is expected to be applicable to various other object-oriented languages and programming logics. A roundup of the results and a discussion of ongoing and future research in Section 6 conclude the paper.

This article is an extended version of [20] which builds loosely on some observations in [17].

2. Denotational model of the object calculus

In this section, we describe a most simple denotational semantics for functional and imperative object calculi within the category PreDom of predomains and partial continuous functions. Let $A \rightharpoonup B$ denote the partial continuous function space between predomains A and B. By $f(a)\uparrow$ we denote that function f applied to a is undefined whereas $f(a)\downarrow$ denotes definedness. Equivalently, one could work within a category of domains (with least elements) and strict functions.

[2] Usually, if Φ is monotonic then S is recursively defined. But monotonicity is too strong a condition for object specifications.

Table 1
Definition of record update and extension

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

Table 2
Definition of record update only

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

If s and t are terms denoting elements of a predomain then we write $s \simeq t$ to state that s and t are strongly equal, i.e. s is defined if, and only if, t is defined and s and t are equal in this case.

2.1. 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:

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

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 a_i $(1 \leqslant i \leqslant k)$ is written $\{\!| l_1 = a_1, \ldots, l_k = a_k |\!\}$. Notice that $\mathsf{Rec}_{\mathbb{L}}(A)$ is always non-empty as it contains the element $\langle \emptyset, \emptyset \rangle$ and that $\mathsf{Rec}_{\mathbb{L}}(A)$ is a flat predomain if A is flat. Thus, in this case, a record and its extension are *in*comparable. Basic record operations like selection and update are defined below.

Definition 2.1. Let $r \in \mathsf{Rec}_{\mathbb{L}}(A)$ such that $r = \langle L, f \rangle$ with $L \subseteq_{\mathsf{fin}} \mathbb{L}$ and $f \in A^L$.

A label l in record $r = \langle L, f \rangle$ is defined, short $l \in \mathsf{dom}\, r$, if, and only if, $l \in L$. Selection of a label $l \in \mathbb{L}$ in record r, short $r.l$, is defined if, and only if, $l \in \mathsf{dom}\, r$ and yields $f(l) \in A$.

The update and extension operation for records is defined as in Table 1. For the semantics of the object calculi we discuss in this paper, however, update is only allowed for existing fields. Therefore, we define a "pure" update in Table 2 which is undefined for labels not defined in the argument record.

2.2. Functional object calculus

The functional object calculus in use is the one of Adabi and Cardelli [1]. As described in [10] there are basically two types of denotational semantics for objects:

Table 3
Syntax of the functional object calculus

$a, b ::=$	x	variable
$\mid$	$[\mathsf{m}_i = \varsigma(x_i)b_i]^{i=1..n}$	object creation
$\mid$	$a.\mathsf{f}$	field select
$\mid$	$a.\mathsf{m}()$	method call
$\mid$	$a.\mathsf{f} := b$	field update
$\mid$	$a.\mathsf{m} \Leftarrow \varsigma(x)b$	method update

a fixed-point semantics, binding the self-object at object creation, and a self-application semantics, establishing this binding at method call. The first dates back to Cardelli and was prominently used in [5], the second was first mentioned in [9].

We will use the latter as it supports the style of specification introduced later. Those specifications are, in turn, inspired by the object formation rule of [2]. Although the former fixed-point semantics works nicely for functional object languages it is not clear to us how it could be made to work in the imperative setup.

2.2.1. Syntax

The syntax of the functional object calculus of Abadi and Cardelli [1] is given in Table 3 where $\mathscr{M}$ and $\mathscr{F}$ be finite sets of method names and field names, respectively. The ς binder was introduced in [1]. It binds a name for the self-object, i.e. the object that is called to execute the method in question. Due to Abadi and Cardelli [1] is also the $\Leftarrow$ notation for method update.

For the sake of simplicity, methods do not have additional arguments. This is not a real restriction as arguments can be encoded by fields.

2.2.2. Semantics

Let BVal denote the flat predomain of basic values like numbers or booleans. The functional object calculus most naturally finds its interpretation within the recursively specified predomain

$$\mathscr{O} = \mathsf{Rec}_{\mathscr{F}}(\mathsf{BVal} + \mathscr{O}) \times \mathsf{Rec}_{\mathscr{M}}(\mathscr{O} \rightharpoonup \mathscr{O})$$

which is non-empty as record types are always non-empty. If we choose BVal to be empty we get the recursive type

$$(\dagger) \qquad \mathscr{O} = \mathsf{Rec}_{\mathscr{F}}(\mathscr{O}) \times \mathsf{Rec}_{\mathscr{M}}(\mathscr{O} \rightharpoonup \mathscr{O}).$$

One can also replace fields by (so-called query-) methods to obtain the most simple recursive type

$$\mathscr{O} = \mathsf{Rec}_{\mathscr{M}}(\mathscr{O} \rightharpoonup \mathscr{O})$$

which strongly reminds one of call-by-value lambda calculus as given by the type equation $L = L \rightharpoonup L$. The difference is essentially that an object is not just a partial continuous function from objects to objects but a whole record of such. For an “object”

Table 4
Semantics of the functional object calculus

$[\![x]\!]\rho$	$=$	$\rho(x)$
$[\![[\mathsf{m}_i = \varsigma(x_i)b_i]^{i=1..n}]\!]\rho$	$=$	$\{\!\vert \mathsf{m}_i = \lambda o.[\![b_i]\!]\rho[o/x_i]\vert\!\}^{i=1..n}$
$[\![a.\mathsf{f}]\!]\rho$	$=$	$([\![a]\!]\rho).\mathsf{f}$
$[\![a.\mathsf{m}()]\!]\rho$	$=$	$([\![a]\!]\rho).\mathsf{m}([\![a]\!]\rho)$
$[\![a.\mathsf{f} := b]\!]\rho$	$=$	$[\![a]\!]\rho[\mathsf{f} := [\![b]\!]]$
$[\![a.\mathsf{m} \Leftarrow \varsigma(x)b]\!]\rho$	$=$	$[\![a]\!]\rho[\mathsf{m} := \lambda o.[\![b]\!]\rho[o/x]]$.

Table 5
Syntax of the imperative object calculus

a, b	$::=$	x	variable
	$\vert$	$[\mathsf{m}_i = \varsigma(x_i)b_i]^{i=1..n}$	object creation
	$\vert$	$a.\mathsf{f}$	field selection
	$\vert$	$a.\mathsf{f} := b$	field update
	$\vert$	$a.\mathsf{m}()$	method call
	$\vert$	$a.\mathsf{m} \Leftarrow \varsigma(x)b$	method update
	$\vert$	$\mathtt{clone}(a)$	shallow copy
	$\vert$	$\mathtt{let}\, x = a\, \mathtt{in}\, b$	local def

$o \in \mathcal{O}$ and a "message" $\mathsf{m} \in \mathcal{M}$ the result of "sending message m to object o" is given by $o.\mathsf{m}(o)$ which is understood as divergent if m does not occur as a label in the record o. It makes sense to conceive methods as partial continuous functions from $\mathcal{O}$ to $\mathcal{O}$ (or total *strict* functions in a category of domains) because if $o.\mathsf{m}$ is defined then the argument o has to be defined as well.

Definition 2.2. We write $[\![a]\!]\rho$ for the interpretation of object expression a in the environment $\rho \in \mathsf{Env} = \mathcal{O}^{\mathsf{Var}}$. This interpretation is defined by structural recursion on object expressions in Table 4. Note that for an $o \in \mathcal{O}$ we write $o.\mathsf{f}$ and $o.\mathsf{m}$ instead of $\pi_1(o).\mathsf{f}$ and $\pi_2(o).\mathsf{m}$, resp., to reduce syntactic clutter. This simplification will be applied throughout the paper.

2.3. Imperative object calculus

The imperative object calculus is more challenging since objects are persistent and reside on the heap. Objects have an identity, usually the location referring to them. Again we follow [1] for syntax and semantics. More exactly, we use a mild variation of the store-model used in [1].

2.3.1. Syntax

The syntax of the imperative untyped object calculus of Abadi and Cardelli [1] is as shown in Table 5 where we distinguish between fields and methods.

Note that sequential composition of commands a and b, short $a; b$, can be expressed as $\mathtt{let}\,\_ = a\ \mathtt{in}\ b$.

2.3.2. Semantics

The imperative object calculus finds its interpretation within the following slightly more complicated system of recursive types:

(1) $\mathsf{Val} = \mathsf{BVal} + \mathsf{Loc}$,
(2) $\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$,
(3) $\mathsf{Ob} = \mathsf{Rec}_{\mathscr{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathscr{M}}(\mathsf{Cl})$,
(4) $\mathsf{Cl} = \mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}$,

where Loc is some countable set of locations (considered as a flat predomain). Some notation will come in handy in later sections. For $\mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathscr{F}}(\mathsf{Val}))$, the part of the store which just contains field values, we write $\mathsf{St}_{\mathsf{Val}}$. There is an obvious projection $\pi_{\mathsf{Val}} : \mathsf{St} \to \mathsf{St}_{\mathsf{Val}}$ given by $\pi_{\mathsf{Val}}(\sigma).\ell \simeq \pi_1(\sigma.\ell)$ where π_1 projects on the first component.

Notice 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. We think that this modelling of states as records should also be employed when modelling, e.g. simple imperative languages where only basic values can be stored in locations. Besides conceptual adequacy a technical advantage of such a modelling is that $\mathsf{Rec}_{\mathsf{Loc}}(\mathbb{N})$ is a countable flat predomain whereas the traditional choice $(\mathsf{Loc} \to \mathbb{N})_\perp$ is flat but not ω-algebraic.

Definition 2.3. Given an environment $\rho \in \mathsf{Env} = \mathsf{Val}^{\mathsf{Var}}$ and an object expression a its interpretation $\llbracket a \rrbracket \rho : \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}$ is defined in Table 6. Again, we write $o.\mathsf{f}$ and $o.\mathsf{m}$ instead of $\pi_1(o).\mathsf{f}$ and $\pi_2(o).\mathsf{m}$, resp., to reduce syntactic clutter.

Note that the "**let** $x = a$ **in** b" used on the right-hand side in Table 6 is a semantical operation on predomains which is undefined should a be undefined.

If one does not distinguish between methods and fields and ignores basic values, see [1], the above system of mutual recursive type definitions simplifies as follows:

(1) $\mathsf{St} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$,
(2) $\mathsf{Ob} = \mathsf{Rec}_{\mathscr{M}}(\mathsf{Cl})$,
(3) $\mathsf{Cl} = \mathsf{Loc} \times \mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St}$.

Table 6
Denotational semantics for the imperative object calculus

$\llbracket x \rrbracket \rho\sigma$	$=$	$\langle \rho(x), \sigma \rangle$
$\llbracket [\mathsf{m}_i = \varsigma(x_i) b_i]^{i=1..n} \rrbracket \rho\sigma$	$=$	$\langle \ell, \sigma[\ell := \{\!\vert \mathsf{m}_i = \lambda\langle \ell', \sigma' \rangle . \llbracket b_i \rrbracket \rho[\ell'/x_i]\, \sigma' \vert\!\}^{i=1...n}] \rangle$
		where ℓ is a fresh location not in the domain of σ
$\llbracket a.\mathsf{f} \rrbracket \rho\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\sigma$ **in** $\langle \sigma'.\ell.\mathsf{f}, \sigma' \rangle$
$\llbracket a.\mathsf{f} := b \rrbracket \rho\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\sigma$ **in** $\langle \ell, \sigma'[\ell := \sigma'.\ell[\mathsf{f} := \llbracket b \rrbracket \rho\, \sigma']] \rangle$
$\llbracket a.\mathsf{m}() \rrbracket \rho\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\sigma$ **in** $\sigma'.\ell.\mathsf{m}(\ell, \sigma')$
$\llbracket a.\mathsf{m} \Leftarrow \varsigma(x) b \rrbracket \rho\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\,\sigma$ **in** $\langle \ell, \sigma'[\ell := \sigma'.\ell[\mathsf{m} := \lambda\langle \ell', \sigma'' \rangle . \llbracket b \rrbracket \rho[\ell'/x]\sigma'']] \rangle$
$\llbracket \mathtt{clone}(a) \rrbracket \rho\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\,\sigma$ **in** $\langle \ell', \sigma'[\ell' := \sigma'.\ell] \rangle$
		where ℓ' is a fresh location not in the domain of σ'
$\llbracket \mathtt{let}\, x = a\, \mathtt{in}\, b \rrbracket \rho\,\sigma$	$=$	**let** $\langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho\sigma$ **in** $\llbracket b \rrbracket \rho[\ell/x]\sigma'$

Notice that equivalently Ob can be defined by the single recursive equation

$$\mathsf{Ob} = \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}) \rightharpoonup \mathsf{Loc} \times \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}))$$

which, obviously, is obtained from $\mathcal{O} = \mathsf{Rec}_{\mathcal{M}}(\mathcal{O} \rightharpoonup \mathcal{O})$ by simply replacing $\mathcal{O}$ by $\mathsf{Loc} \times \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob})$ on the right-hand side.

2.3.3. Variation à la Abadi and Cardelli

The denotational semantics presented in Section 2.3.2 is not quite in accordance with the operational semantics for the imperative object calculus given in the book [11, pp. 136–137] which insinuates the following domain equations:

$$\begin{array}{lll}(1) & \mathsf{Val} & = \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc}),\\ (2) & \mathsf{St} & = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Cl}),\\ (3) & \mathsf{Cl} & = \mathsf{Val} \times \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St},\end{array}$$

where method closures are saved directly in the store and meanings of object expressions a in an environment $\rho \in \mathsf{Env} = \mathsf{Val}^{\mathsf{Var}}$ are functions of type $\mathsf{St} \rightharpoonup \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc}) \times \mathsf{St}$ and not of type $\mathsf{St} \rightharpoonup \mathsf{Loc} \times \mathsf{St}$.

3. Object specifications

Having identified the meaning of the (functional and imperative resp.) object calculus (within the recursively defined predomain $\mathcal{O}$ and $\mathsf{Loc} \times \mathsf{St}$) we are in the position to use any logic of predomains for reasoning about objects. One might find it useful to identify a special purpose calculus[3] for reasoning about objects which finds its meaning by translation into some logic of predomains. However, before embarking on such a project we discuss what is the shape of existing predicates expressing interesting properties of objects.

To that end, we consider an example. Let $o = [\![a]\!]$ be the object representing a point with a method computing the distance from the origin which is wrapped inside an object

$$a = [x = 10, y = 0, \mathsf{dist} = \varsigma(o)[\mathsf{res} = \mathsf{sqrt}(o.x^2 + o.y^2)]].$$

This object may be specified by a predicate requiring

- the fields x and y to satisfy $o.x \in \mathbb{N} \wedge o.y \in \mathbb{N}$;
- the result o' of the method dist to satisfy $o'.\mathsf{res} \in \mathbb{N}$;
- the relation between input o and output o' of method dist to satisfy $o'.\mathsf{res} = \mathsf{sqrt}(o.x^2 + o.y^2)$.

In general, there are three ingredients to any specification: a predicate A to denote the specification of fields, a predicate B_{m} for the result specification of method m, and

[3] As e.g. Hoare logic which provides a useful "macro-language" for reasoning about partial functions on states.

a relation T_{m} for the input/output (or transition) specification of method m. These predicates can be put together in different ways to yield a notion of specification as described informally in the example above. Two such possible definitions are discussed below. Again, we use the functional object-calculus for presentation as it is technically simpler.

Definition 3.1. Let $A \in \mathscr{P}(\mathcal{O}) \to \mathscr{P}(\mathcal{O})$ and $\vec{B} = (B_{\mathsf{m}} \in \mathscr{P}(\mathcal{O}) \to \mathscr{P}(\mathcal{O}))_{\mathsf{m}\in\mathscr{M}}$ such that A and B_{m} are monotonic w.r.t. $\subseteq$ for all $m \in \mathscr{M}$, and $\vec{T} = (T_{\mathsf{m}} \subseteq \mathcal{O} \times \mathcal{O})_{\mathsf{m}\in\mathscr{M}}$. Then these data induce a monotonic operator $\Phi_{A,\vec{B},\vec{T}} : \mathscr{P}(\mathcal{O}) \to \mathscr{P}(\mathcal{O})$ which is defined as

$$o \in \Phi_{A,\vec{B},\vec{T}}(S) \equiv o \in A(S) \wedge \\ \forall \mathsf{m} \in \mathscr{M}. o.\mathsf{m}(o)\downarrow \Rightarrow o.\mathsf{m}(o) \in B_{\mathsf{m}}(S) \wedge T_{\mathsf{m}}(o, o.\mathsf{m}(o))$$

for $S \in \mathscr{P}(\mathcal{O})$. The B_{m} stand for result specifications and the T_{m} represent transition specifications for each method m. Finally, A specifies the remaining properties of the object, i.e. the fields. We write $\mathsf{Inv}(A, \vec{B}, \vec{T})$ for the *greatest* fixpoint of $\Phi_{A,\vec{B},\vec{T}}$.

If I is a post-fixpoint of $\Phi_{A,\vec{B},\vec{T}}$, i.e. $I \subseteq \Phi_{A,\vec{B},\vec{T}}(I)$, then every $o \in I$ satisfies the predicate A and whenever $o.\mathsf{m}(o)$ is defined then it satisfies $B_{\mathsf{m}}(I)$ and is related to o via T_{m}. In particular, this holds for the greatest fixpoint $\mathsf{Inv}(A, \vec{B}, \vec{T})$ of $\Phi_{A,\vec{B},\vec{T}}$ as given by $\bigcup\{I \in \mathscr{P}(\mathcal{O}) \mid I \subseteq \Phi_{A,\vec{B},\vec{T}}(I)\}$, the union of all post-fixpoints of $\Phi_{A,\vec{B},\vec{T}}$. Thus, in order to prove that $o \in \mathsf{Inv}(A, \vec{B}, \vec{T})$ it suffices to exhibit a predicate P with $P \subseteq \Phi_{A,\vec{B},\vec{T}}(P)$ and $o \in P$.

Such a notion of "invariant" specification seems to be quite in accordance with the "coalgebraic view" of the object-oriented world and, therefore, is probably quite useful. However, it seems to have its limitations as exemplified by the following example.

Example 3.1. Consider the object expression $a \equiv [\mathsf{m} = \varsigma(x)x.\mathsf{m}()]$. Operational intuition tells us that $a.\mathsf{m}()$ diverges and, therefore, it would be most desirable to prove this employing an appropriate notion of invariant. What immediately comes to mind is the invariant $I = \mathsf{Inv}(A, \vec{B}, \vec{T})$ with $A(S)(x) \equiv \mathit{True}$, $T_{\mathsf{m}}(x, y) \equiv \mathit{False}$ and $B_{\mathsf{m}}(S) \equiv S$. Then for I we have

$$o \in I \Leftrightarrow \neg o.\mathsf{m}(o)\downarrow .$$

Coinduction does not help in proving that $[\![a]\!] \in I$ for an object a since one has to find a predicate P such that $o \in P \Rightarrow o.\mathsf{m}(o)\downarrow \Rightarrow \mathit{False}$. But since the specification does not contain I, the only canonical choice for P is I again, so nothing is achieved.

In [2] an axiomatic logic was introduced for a variant of the imperative object calculus which allows one to prove divergence of $a.\mathsf{m}()$ quite easily. For sake of simplicity, we first discuss the following adaptation of their account to the purely functional case.

3.1. Functional object specifications

A notion of specification for functional objects is suggested. The existence of such specifications is discussed in Section 4.

Definition 3.2. Given $A \in \mathscr{P}(\mathscr{O}) \to \mathscr{P}(\mathscr{O})$, $\vec{B} = (B_{\mathsf{m}} \in \mathscr{P}(\mathscr{O}) \to \mathscr{P}(\mathscr{O}))_{\mathsf{m}\in\mathscr{M}}$ and $\vec{T} = (T_{\mathsf{m}} \in \mathscr{P}(\mathscr{O} \times \mathscr{O}))_{\mathsf{m}\in\mathscr{M}}$, let $\mathsf{Spec}(A, \vec{B}, \vec{T})$ be the predicate $S \subseteq \mathscr{O}$ with

$$\begin{aligned} o \in S \equiv o \in A(S) \wedge \\ \forall \mathsf{m} \in \mathscr{M}.\ \forall o' \in S.\ o.\mathsf{m}(o')\downarrow \Rightarrow o.\mathsf{m}(o') \in B_{\mathsf{m}}(S) \wedge T_{\mathsf{m}}(o', o.\mathsf{m}(o')) \end{aligned}$$

provided S is *unique* with this property. We call $\mathsf{Spec}(A, \vec{B}, \vec{T})$ the *specification* induced by A, $\vec{B}$ and $\vec{T}$.

This is different from $\mathsf{Inv}(A, \vec{B}, \vec{T})$ since one requires for methods $\mathsf{m} \in \mathscr{M}$ the condition $\forall o' \in S.\ o.\mathsf{m}(o')\downarrow \Rightarrow o.\mathsf{m}(o') \in B_{\mathsf{m}}(S) \wedge T_{\mathsf{m}}(o', o.\mathsf{m}(o'))$ to hold and not just $o.\mathsf{m}(o)\downarrow \Rightarrow o.\mathsf{m}(o) \in B_{\mathsf{m}}(S) \wedge T_{\mathsf{m}}(o, o.\mathsf{m}(o))$. Note that S is implicitly ("recursively") specified even if the $\vec{B}$ and A do not depend on S.

Example 3.2. To illustrate this new notion we will employ the specification $S = \mathsf{Spec}(\mathrm{True}, \mathrm{True}, \mathrm{False})$ satisfying

$$o \in S \quad \Leftrightarrow \quad \forall o' \in S.\ o.\mathsf{m}(o')\downarrow \Rightarrow \mathit{False}$$

(i.e. $\forall o' \in S.\ o.\mathsf{m}(o')\uparrow$) for showing $[\![a.\mathsf{m}()]\!]\uparrow$ for $a \equiv [\mathsf{m} = \varsigma(x)x.\mathsf{m}()]$ from Example 3.1 above.

Of course, from $o \in S$ it follows that $o.\mathsf{m}(o)\uparrow$. Thus, it remains to show that $[\![a]\!] \in S$ which, however, is easily seen to be the case as for $o' \in S$ we have $[\![a]\!].\mathsf{m}(o') = o'.\mathsf{m}(o')$ which diverges by the previous consideration.

3.2. Imperative object specifications

For the imperative setting the corresponding notion of specification is obtained analogously to the functional case yet accounting for the underlying store (the different "implementation" of $\mathscr{O}$). Again, existence of such specifications is discussed in Section 4. As before, the predicates A, B_{m}, and T_{m} denote field specification, method result specification, and method transition specification, respectively. In the imperative case, however, they have other types since $\mathscr{O}$ becomes $\mathsf{Loc} \times \mathsf{St}$.

Definition 3.3. For any predicates or families of predicates, resp.,

$$\begin{aligned} & A \in \mathscr{P}(\mathsf{Loc} \times \mathsf{St}) \to \mathscr{P}(\mathsf{Loc} \times \mathsf{St}), \\ & \vec{B} = (B_{\mathsf{m}} \in \mathscr{P}(\mathsf{Loc} \times \mathsf{St}) \to \mathscr{P}(\mathsf{Val} \times \mathsf{St}))_{\mathsf{m}\in\mathscr{M}}, \\ & \vec{T} = (T_{\mathsf{m}} \in \mathscr{P}(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}))_{\mathsf{m}\in\mathscr{M}}, \end{aligned}$$

let $\mathsf{Spec}(A, \vec{B}, \vec{T})$ be the predicate $S \subseteq \mathsf{Loc} \times \mathsf{St}$ with

$$\begin{aligned}\langle \ell, \sigma\rangle \in S \equiv A(S)(\ell,\sigma) \wedge \\ &\forall \mathsf{m} \in \mathscr{M}.\ \forall \ell' \in \mathsf{Loc}.\ \forall \sigma' \in \mathsf{St}.\ \langle \ell', \sigma'\rangle \in S \Rightarrow \\ &\forall v \in \mathsf{Val}.\ \forall \sigma'' \in \mathsf{St}. \\ &\qquad \sigma.\ell.\mathsf{m}(\ell', \sigma') = \langle v, \sigma''\rangle \Rightarrow B_{\mathsf{m}}(S)(v, \sigma'') \wedge T_{\mathsf{m}}(\ell', \sigma', v, \sigma'')\end{aligned}$$

provided S is *unique* with the above property.

If $\vec{T} = (T_{\mathsf{m}} \in \mathscr{P}(\mathsf{Loc} \times \mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}}))_{\mathsf{m} \in \mathscr{M}}$ let $\mathsf{Spec}_{\mathsf{flat}}(A, \vec{B}, \vec{T})$ be the predicate $S \subseteq \mathsf{Loc} \times \mathsf{St}$ with

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

provided S is *unique* with this property.

In Section 4 it will become clear why it is useful to restrict attention to transition specifications that just refer to the "flat" part of the store and not to the "higher-order part" of the store, i.e. the method closures.

Example 3.3. Assume that object specification S is supposed to express that field f is a natural number greater than zero, that method m returns an object that again satisfies S and that this method does not decrease the value of f. Define S accordingly:

$$\begin{aligned}A(S)(\ell,\sigma) &\equiv \sigma.\ell.\mathsf{f} \in \mathbb{N} \wedge \sigma.\ell.\mathsf{f} > 0, \\ T_{\mathsf{m}}(\ell,\sigma,v,\sigma') &\equiv \sigma.\ell.\mathsf{f} \in \mathbb{N} \Rightarrow \sigma'.\ell.\mathsf{f} \in \mathbb{N} \wedge \sigma'.\ell.\mathsf{f} \geqslant \sigma.\ell.\mathsf{f}, \\ B_{\mathsf{m}}(S) &\equiv S.\end{aligned}$$

Note that B_{m} is recursive. It requires that specification S also holds for the result of m. The object $[\mathsf{f} = 12, \mathsf{m} = \varsigma(x)x]$ would thus fulfil the specification S as would $[\mathsf{f} = 12, \mathsf{m} = \varsigma(x)x.\mathsf{f} := x.\mathsf{f} + 1; x]$.

Despite their indisputable usefulness, the problem with specifications is, however, that there is no obvious reason why they should exist as the right-hand side of the equivalence characterising $\mathsf{Spec}(A, \vec{B}, \vec{T})$ contains both positive and negative occurrences of $\mathsf{Spec}(A, \vec{B}, \vec{T})$. Though in [2] specifications are used intrinsically their existence is not verified. Instead the validity of assertions for programs is defined w.r.t. *derivability of correctness assertions* which renders the value of the Soundness Theorem of Abadi and Leino [2] as somewhat mysterious.

4. Existence of object specifications

In this section, we will identify some mild assumptions which guarantee the existence and uniqueness of the specifications introduced in the previous section.

A particular kind of predicates will be needed, *admissible predicates*. These are, as usual, predicates preserved by suprema of ascending chains.

4.1. Functional object specifications

In contrast to functional[4] or imperative kernel languages the object calculus implicitly presupposes recursive types like $\mathcal{O}$. Thus, it appears necessary to employ induction principles for the recursive type involved in order to verify programs. After recalling in concrete terms the induction principle for $\mathcal{O}$ we will use it to establish the existence of specifications under fairly mild conditions. For the sake of presentation, let us work with the simpler domain equation $\mathcal{O} = \mathsf{Rec}_{\mathcal{M}}(\mathcal{O} \rightharpoonup \mathcal{O})$ where we do not distinguish between fields and methods.

From well-known work of Freyd and Pitts in the early 1990s [7,14] we know that the *bifree* solutions of the domain equation $A = F(A, A)$ can be characterised by the requirement that id_A is the least fixpoint of $\delta_F = \lambda e.F(e, e)$. Note that we deal with domain equations up to equality. In case of $F(Y, X) = \mathsf{Rec}_{\mathcal{M}}(Y \rightharpoonup X)$ we write δ for δ_F which is defined explicitly as the endo-function on $[\mathcal{O} \rightharpoonup \mathcal{O}]$ as given by

$$\delta(e)(\{\!|\mathsf{m}_i = f_i|\!\}^{i=1..n}) = \{\!|\mathsf{m}_i = e \circ f_i \circ e|\!\}^{i=1..n}$$

or, equivalently, in a more readable form by

$$\delta(e)(a).\mathsf{m} = e \circ a.\mathsf{m} \circ e$$

for $e : \mathcal{O} \rightharpoonup \mathcal{O}$, $a \in \mathcal{O}$ and $\mathsf{m} \in \mathcal{M}$.

From $\mathsf{id} = \mu(\delta) = \bigsqcup_{n \in \mathbf{N}} \delta^n(\bot)$ it follows immediately that $P(\mathsf{id})$ holds for an admissible predicate $P \subseteq [\mathcal{O} \rightharpoonup \mathcal{O}]$ if $P(\bot)$ and $\forall e \sqsubseteq \mathsf{id}.\, P(e) \Rightarrow P(\delta(e))$. This *fixpoint induction* principle can be used directly for verifying properties of objects.

Example 4.1. Let $a = [\mathsf{m} = \varsigma(x)x.\mathsf{m}()]$, then using Fixpoint Induction one can prove that $[\![a.\mathsf{m}()]\!]\!\uparrow$.

Let $o = [\![a]\!]$ and consider the admissible predicate

$$P(e) \equiv e(o).\mathsf{m}(e(o))\!\uparrow$$

on $[\mathcal{O} \rightharpoonup \mathcal{O}]$. Obviously, $[\![a.\mathsf{m}()]\!]\!\uparrow$ is equivalent to $P(\mathsf{id})$. Thus, by fixpoint induction it suffices to show that $\forall e \sqsubseteq \mathsf{id}.P(e) \Rightarrow P(\delta(e))$. Suppose that $e \sqsubseteq \mathsf{id}$ with $P(e)$,

[4] For example PCF is based on the finite type hierarchy over the base type $\mathbb{N}_\bot$ and simple imperative languages for which Hoare calculus was first introduced are based on $\mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Val}) \rightharpoonup \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Val})$.

i.e., $e(o).\mathsf{m}(e(o))\uparrow$. Then $P(\delta(e))$ as

$$\begin{aligned}\delta(e)(o).\mathsf{m}(\delta(e)(o)) &\sqsubseteq \delta(e)(o).\mathsf{m}(o)\\ &= e(o.\mathsf{m}(e(o)))\\ &\sqsubseteq o.\mathsf{m}(e(o))\\ &= e(o).\mathsf{m}(e(o))\uparrow,\end{aligned}$$

where the last equality is the induction hypothesis $P(e)$.

The fixpoint induction principle will be employed once more below for proving unique existence of specifications under rather mild assumptions.

Definition 4.1. For a flat predomain I let $\mathscr{L}_I(A)$ be the complete lattice of admissible subsets of $I \times A$ ordered by $\subseteq$.

Let I be a flat predomain. For any $X, Y \in \mathscr{L}_I(A)$, and $e \in [A \rightharpoonup A]$ we define

$$e : X \subseteq Y \equiv \forall \ell \in I.\ \forall a \in A. \langle \ell, a\rangle \in X \wedge e(a)\downarrow \Rightarrow \langle \ell, e(a)\rangle \in Y$$

as in [14].

For $X, Y \in \mathscr{L}_I(A)$ the set $\{e \in A \rightharpoonup A \mid e : X \subseteq Y\}$ is obviously a non-empty Scott-closed subset of the domain $[A \rightharpoonup A]$.

The following theorem uses the same line of arguments as [14].

Theorem 4.2. *Given a locally continuous bifunctor on predomains F and a predomain A which is a bifree solution of F, $A = F(A, A)$, a predomain I, and a monotonic $\Phi : \mathscr{L}_I(A)^{op} \times \mathscr{L}_I(A) \to \mathscr{L}_I(A)$, such that*

$$(\dagger) \quad e : X \subseteq X' \wedge e : Y' \subseteq Y \Rightarrow F(e, e) : \Phi(Y, X) \subseteq \Phi(Y', X')$$

for all $X, Y, X', Y' \in \mathscr{L}_I(A)$ and $e \sqsubseteq \mathsf{id}_A$.

Then $S = \Phi(S, S)$ for a unique *$S \in \mathscr{L}_I(A)$.*

Proof. Let $\Phi : \mathscr{L}_I(A)^{op} \times \mathscr{L}_I(A) \to \mathscr{L}_I(A)$ be monotonic and satisfy the condition ($\dagger$). Then the mapping

$$\begin{aligned}&\hat{\Phi} : \mathscr{L}_I(A)^{op} \times \mathscr{L}_I(A) \to \mathscr{L}_I(A)^{op} \times \mathscr{L}_I(A)\\ &\hat{\Phi}(Y, X) \mapsto (\Phi(X, Y), \Phi(Y, X))\end{aligned}$$

is a monotonic endomap on the complete lattice $\mathscr{L}_I(A)^{op} \times \mathscr{L}_I(A)$. Thus, by Knaster–Tarski $\hat{\Phi}$ has a fixpoint $(S^-, S^+) = \hat{\Phi}(S^-, S^+)$.

For establishing $S^- = S^+$ we show by fixpoint induction that for the admissible predicate

$$P(e) \equiv e \sqsubseteq \mathsf{id}_A \wedge e : S^- \subseteq S^+ \wedge e : S^+ \subseteq S^-$$

we have $P(\mu e.F(e, e))$ and, therefore, $P(\mathsf{id}_A)$ as $\mathsf{id}_A = \mu e.F(e, e)$ from which it follows that $S^- = S^+$. Obviously, we have $P(\perp)$. For the induction step assume that

$P(e)$. Then $F(e,e) \sqsubseteq F(\mathsf{id},\mathsf{id}) = \mathsf{id}$. Moreover, from $e : S^- \subseteq S^+$ it follows by (†) that $F(e,e) : S^- = \Phi(S^+,S^-) \subseteq \Phi(S^-,S^+) = S^+$ and, analogously, it follows from $e : S^+ \subseteq S^-$ by (†) that $F(e,e) : S^+ = \Phi(S^-,S^+) \subseteq \Phi(S^+,S^-) = S^-$. Thus, we have $P(F(e,e))$.

Thus, we conclude that there exists at least one $S \in \mathscr{L}_I(A)$ with $S = \Phi(S,S)$. For showing uniqueness suppose $S' = \Phi(S',S')$ for some $S' \in \mathscr{L}_I(A)$. For the admissible predicate

$$P(e) \equiv e \sqsubseteq \mathsf{id}_A \wedge e : S \subseteq S' \wedge e : S' \subseteq S,$$

it follows that $P(\mu e.F(e,e))$ again by fixpoint induction. Obviously, we have $P(\bot)$. Assume that $P(e)$. Then $F(e,e) \sqsubseteq F(\mathsf{id},\mathsf{id}) = \mathsf{id}$. Moreover, it follows by (†) that

$$F(e,e) : S = \Phi(S,S) \subseteq \Phi(S',S') = S',$$
$$F(e,e) : S' = \Phi(S',S') \subseteq \Phi(S,S) = S$$

as the induction hypothesis $P(e)$ ensures $e : S \subseteq S'$ and $e : S' \subseteq S$. But as $\mathsf{id} = \mu e.F(e,e)$ we have $P(\mathsf{id})$ from which it follows immediately that $\mathsf{id} : S \subseteq S'$ and $\mathsf{id} : S' \subseteq S$, i.e. $S \subseteq S'$ and $S' \subseteq S$, and, therefore $S = S'$ as desired. □

Theorem 4.3 (Existence theorem). *Let $\mathscr{L}$ denote $\mathscr{L}_1(\mathcal{O})$ and $F(Y,X) = \mathsf{Rec}_{\mathscr{F}}(X) \times \mathsf{Rec}_{\mathscr{M}}(Y \rightharpoonup X)$. Moreover, let $A \in \mathscr{L} \to \mathscr{L}$, $\vec{B} = (B_{\mathsf{m}} \in \mathscr{L} \to \mathscr{L})_{\mathsf{m} \in \mathscr{M}}$ and $\vec{T} = (T_{\mathsf{m}} \in \mathscr{P}(\mathcal{O} \times \mathcal{O}))_{\mathsf{m} \in \mathscr{M}}$ be families such that for all $\mathsf{m} \in \mathscr{M}$*

(1) $e : X \subseteq Y$ implies $F(e,e) : A(X) \subseteq A(Y)$ *for $e \sqsubseteq \mathsf{id}_{\mathcal{O}}$ and $X, Y \in \mathscr{L}$;*
(2) $e : X \subseteq Y$ *implies* $e : B_{\mathsf{m}}(X) \subseteq B_{\mathsf{m}}(Y)$ *for $e \sqsubseteq \mathsf{id}_{\mathcal{O}}$ and $X, Y \in \mathscr{L}$;*
(3) $T_{\mathsf{m}}(o,-) := \{o' \in \mathcal{O} \mid T_{\mathsf{m}}(o,o')\}$ *is Scott-closed for all $o \in \mathcal{O}$ and $T_{\mathsf{m}}(o,-) \subseteq T_{\mathsf{m}}(o',-)$ whenever $o \sqsubseteq o'$.*

Then there exists a unique $S \in \mathscr{L}$ satisfying for all $o \in \mathcal{O}$

$$(*) \quad o \in S \equiv o \in A(S) \wedge \\ \forall \mathsf{m} =\in \mathscr{M}.\ \forall o' \in S.\ o.\mathsf{m}(o')\downarrow \Rightarrow o.\mathsf{m}(o') \in B_{\mathsf{m}}(S) \wedge T_{\mathsf{m}}(o', o.\mathsf{m}(o')).$$

Proof. For $Y, X \in \mathscr{L}$ consider the predicate

$$o \in \Phi(Y,X) \equiv o \in A(X) \wedge \\ \forall \mathsf{m} =\in \mathscr{M}.\ \forall o' \in Y.\ o.\mathsf{m}(o')\downarrow \Rightarrow o.\mathsf{m}(o') \in B_{\mathsf{m}}(X) \\ \wedge T_{\mathsf{m}}(o', o.\mathsf{m}(o'))$$

which is admissible if X and Y are due to the fact that B_{m} and $T_{\mathsf{m}}(o,-)$ are admissible (see also condition (3)) and that the precondition of the implication is downward-closed. Clearly, the operator $\Phi : \mathscr{L}^{op} \times \mathscr{L} \to \mathscr{L}$ is monotonic as A and B are by (1) and (2).

Obviously, the requirement $S = \Phi(S,S)$ is equivalent to $(*)$ for all $o \in \mathcal{O}$. Thus, we have to show that there exists a unique $S \in \mathcal{L}$ with $S = \Phi(S,S)$ which is guaranteed by Theorem 4.2 provided we can show that our Φ satisfies the condition (†) of Theorem 4.2 which we verify next.

Suppose $e \sqsubseteq \mathsf{id}_{\mathcal{O}}$ with $e : X \subseteq X'$ and $e : Y' \subseteq Y$.

For showing $F(e,e) : \Phi(Y,X) \subseteq \Phi(Y',X')$ suppose $o \in \Phi(Y,X)$ and show that $F(e,e)(o) \in \Phi(Y',X')$.

First we show that $F(e,e)(o) \in A(X')$. But $F(e,e)(o) \sqsubseteq o \in A(X)$ and, therefore, also $F(e,e)(o) \in A(X')$ due to assumption (1).

Next, let $\mathsf{m} \in \mathcal{M}$ and $o' \in Y'$ with $F(e,e)(o).\mathsf{m}(o')\downarrow$. We then get $e(o.\mathsf{m}(e(o')))\downarrow$ from $F(e,e)(o).\mathsf{m}(o') = e(o.\mathsf{m}(e(o')))$ and thus also $o.\mathsf{m}(e(o'))\downarrow$ and $e(o')\downarrow$. Then $e(o') \in Y$ as $e : Y' \subseteq Y$ and, therefore, as by induction hypothesis $o \in \Phi(Y,X)$), it follows that

$$o.\mathsf{m}(e(o')) \in B_{\mathsf{m}}(X) \wedge T_{\mathsf{m}}(e(o'), o.\mathsf{m}(e(o'))).$$

But then we have

$$e(o.\mathsf{m}(e(o'))) \in B_{\mathsf{m}}(X')$$

by (2) and the assumption $e : X \subseteq X'$. Moreover, we obtain

$$T_{\mathsf{m}}(o', e(o.\mathsf{m}(e(o'))))$$

as $e(o.\mathsf{m}(e(o'))) \sqsubseteq o.\mathsf{m}(e(o'))$, $e(o') \sqsubseteq o'$ and (3) implies that $x' \sqsubseteq x \wedge y \sqsubseteq y'$ implies $T(y,x) \Rightarrow T(y',x')$. Thus, it follows that

$$F(e,e)(o).\mathsf{m}(o') \in B_{\mathsf{m}}(X') \wedge T_{\mathsf{m}}(o', F(e,e)(o).\mathsf{m}(o'))$$

which completes the proof. □

4.2. *Imperative object specifications*

Recall from Section 2 that the imperative object calculus of Abadi and Cardelli [1] finds its denotational interpretation within the recursively defined predomain $\mathsf{St} = F_{\mathsf{St}}(\mathsf{St},\mathsf{St})$ if the latter is defined to be

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

where $\mathsf{Val} = \mathsf{BVal} + \mathsf{Loc}$.

Next, we prove a variant of Theorem 4.3 for the imperative object calculus.

4.2.1. *The imperative existence theorem*

Theorem 4.4. *For any predicates and families of predicates, resp.,*

$$A \in \mathcal{L}_{\mathsf{Loc}}(\mathsf{St}) \to \mathcal{L}_{\mathsf{Loc}}(\mathsf{St}),$$
$$\vec{B} = (B_{\mathsf{m}} \in \mathcal{L}_{\mathsf{Loc}}(\mathsf{St}) \to \mathcal{L}_{\mathsf{Val}}(\mathsf{St}))_{\mathsf{m} \in \mathcal{M}},$$
$$\vec{T} = (T_{\mathsf{m}} \in \mathcal{P}(\mathsf{Loc} \times \mathsf{St} \times \mathsf{Val} \times \mathsf{St}))_{\mathsf{m} \in \mathcal{M}}$$

such that

(i) $e : X \subseteq X'$ *implies* $F_{\mathsf{St}}(e,e) : A(X) \subseteq A(X')$ *for all* $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$;

(ii) *for all* $\mathsf{m} \in \mathscr{M}$, $e : X \subseteq X'$ *implies* $e : B_{\mathsf{m}}(X) \subseteq B_{\mathsf{m}}(X')$ *for all* $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$;

(iii) *for all* $m \in \mathscr{M}$ *the predicate* T_{m} *is Scott-closed in its fourth argument and monotonic in its second argument.*

Then for $\Phi : \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})^{op} \times \mathscr{L}_{\mathsf{Loc}}(\mathsf{St}) \to \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})$ *with*

$$\begin{aligned} \Phi(Y,X)(\ell,\sigma) \equiv A(X)(\ell,\sigma) \wedge \\ & \forall \mathsf{m} \in \mathscr{M}.\ \forall \ell' \in \mathsf{Loc}.\ \forall \sigma' \in \mathsf{St}.\ \langle \ell', \sigma' \rangle \in Y \Rightarrow \\ & \forall v \in \mathsf{Val}.\ \forall \sigma'' \in \mathsf{St}. \\ & \sigma.\ell.\mathsf{m}(\ell',\sigma') = \langle v, \sigma'' \rangle \Rightarrow B_{\mathsf{m}}(X)(v,\sigma'') \wedge T_{\mathsf{m}}(\ell',\sigma',v,\sigma'') \end{aligned}$$

there exists a unique $S \in \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})$ *with* $S = \Phi(S,S)$.

Proof. Instantiating Theorem 4.2 by F_{St} for F, Loc for I, and St for A guarantees the existence of a unique fixpoint for Φ provided we can verify that Φ satisfies the condition (†) of Theorem 4.2.

First, observe that $\Phi(Y,X)$ is admissible w.r.t. σ if X, Y are. This follows from the general fact that if predicate P is open and Q admissible then $P \Rightarrow Q$ is admissible. Also, the operator $\Phi : \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})^{op} \times \mathscr{L}_{\mathsf{Loc}}(\mathsf{St}) \to \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})$ is monotonic by (i) and (ii).

For subsequent use it is helpful to recall that $F_{\mathsf{St}}(e,e)(\sigma)\downarrow$ for all $\sigma \in \mathsf{St}$ and $e : \mathsf{St} \rightharpoonup \mathsf{St}$, and that

(a) $F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{f} = \sigma.\ell.\mathsf{f}$ for all $\mathsf{f} \in \mathscr{F}$ and

(b) $F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{m} = (\mathsf{id}_{\mathsf{Val}} \times e) \circ (\sigma.\ell.\mathsf{m}) \circ (\mathsf{id}_{\mathsf{Loc}} \times e)$ for all $\mathsf{m} \in \mathscr{M}$.

Now we show that Φ satisfies condition (†). Suppose $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$ with

(1) $e : X \subseteq X'$,

(2) $e : Y' \subseteq Y$

for some $X, X', Y, Y' \in \mathscr{L}_{\mathsf{Loc}}(\mathsf{St})$.

We have to show that $F_{\mathsf{St}}(e,e) : \Phi(Y,X) \subseteq \Phi(Y',X')$. For that purpose we suppose that

(3) $\langle \ell, \sigma \rangle \in \Phi(Y,X)$

and show that $\langle \ell, F_{\mathsf{St}}(e,e)(\sigma) \rangle \in \Phi(Y',X')$.

From (3) we get $A(X)(\ell,\sigma)$. Thus by (i) we get that $A(X')(\ell, F_{\mathsf{St}}(e,e)(\sigma))$, i.e. the first part of the conjunction $\langle \ell, F_{\mathsf{St}}(e,e)(\sigma) \rangle \in \Phi(Y',X')$.

For the second part suppose that

(4) $\langle \ell', \sigma' \rangle \in Y'$ with $F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{m}(\ell',\sigma')\downarrow$.

From (b) and $F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{m}(\ell',\sigma')\downarrow$ we know that

(5) $F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{m}(\ell',\sigma') = \langle v, e(\sigma'') \rangle$ with

(6) $\langle v, \sigma'' \rangle = \sigma.\ell.\mathsf{m}(\ell', e(\sigma'))$

for some value $v \in \mathsf{Val}$ and some store $\sigma'' \in \mathsf{St}$. We have to show that

$$B_{\mathsf{m}}(X')(v, e(\sigma'')) \wedge T_{\mathsf{m}}(\ell', \sigma', v, e(\sigma'')).$$

From (6) it follows that $e(\sigma')\downarrow$. Thus, from (4) we get by (2) that

(7) $\langle \ell', e(\sigma') \rangle \in Y$.

Thus, by (3) it follows that $B_{\mathsf{m}}(X)(\sigma.\ell.\mathsf{m}(\ell', e(\sigma')))$, i.e. $B_{\mathsf{m}}(X)\langle v, \sigma''\rangle$ by (6). By (ii) it now follows that $B_{\mathsf{m}}(X')(v, e(\sigma''))$ and, therefore, by (5) that
(8) $B_{\mathsf{m}}(X')(F_{\mathsf{St}}(e,e)(\sigma).\ell.\mathsf{m}(\ell', \sigma'))$.
It follows by the second part of the conjunction $\langle \ell, \sigma\rangle \in \Phi(Y, X)$ as ensured by (3) that
(9) $T_{\mathsf{m}}(\ell', e(\sigma'), v, \sigma'')$
as $\langle \ell', e(\sigma')\rangle \in Y$ by (7) and $\sigma.\ell.\mathsf{m}(\ell', e(\sigma'))\downarrow$ by (6). From $e \sqsubseteq \mathsf{id}_{\mathsf{St}}$ one gets $e(\sigma') \sqsubseteq \sigma'$ and $e(\sigma'') \sqsubseteq \sigma''$. Therefore, by assumption (iii) it follows that
(10) $T_{\mathsf{m}}(\ell', \sigma', v, e(\sigma''))$,
i.e. $\langle \ell', F(e,e)(\sigma')\rangle \in \Phi(Y', X')$. □

This proves that under certain conditions the specification $\mathsf{Spec}(A, \vec{B}, \vec{T})$ exists. But condition (iii) of Theorem 4.4 is awkward to prove and may be replaced by simpler sufficient conditions.

4.2.2. *Possible simplifications of the existence theorem*

If the method specifications T_{m} meet certain requirements Theorem 4.4 becomes less complicated, or more precisely, condition (iii) becomes vacuous.

Corollary 4.5. *Should the T_{m} only refer to the flat part of the store, i.e. $T_{\mathsf{m}}(\ell, \sigma', v, \sigma'') \Leftrightarrow \tilde{T}_{\mathsf{m}}(\ell, \pi_{\mathsf{Val}}(\sigma'), v, \pi_{\mathsf{Val}}(\sigma''))$ then condition* (iii) *of Theorem* 4.4 *becomes vacuously true.*

Proof. This follows simply from the fact that $\mathsf{St}_{\mathsf{Val}}$ is a flat predomain. □

Methods are specified in terms of their result and the state *change* they provoke, in other words, by means of result specification B_{m} and transition specifications T_{m}. If the above corollary is used to ensure existence of specifications it seems impossible to refer to other methods in a transition specification. Such reference is, however, necessary, to specify method transformers, i.e. methods that change or transform methods of other objects (be it the self-object or another one). Such transformer methods are as useful to object-oriented programming as higher-order functions to functional programming.

In order to deal with this problem we consider a way to express properties of other method closures in pre- and postconditions. The canonical choice is to use Hoare-triples:

Definition 4.2. Let an input/output specification of a closure $h \in \mathsf{Cl}$ be defined as follows:

$$\{P\}h\{Q\} = \forall \ell \in \mathsf{Loc}.\ \forall \sigma \in \mathsf{St}.\ P(\ell, \sigma) \wedge h(\ell, \sigma)\downarrow \Rightarrow Q(v, h(\ell, \sigma)),$$

where $P, Q \subseteq \mathsf{Loc} \times \mathsf{St}$.

Lemma 4.6. *If Q is downward-closed* (*Scott-closed resp.*) *then $\{P\}(-)\{Q\}$ is downward-closed* (*Scott-closed resp.*).

Proof. Analogous to the proof of Scott-closedness in Theorem 4.4. □

Note that P does *not* have to be Scott-open as it does not involve the method closure at all.

For example, method specification may depend on the (specified) behaviour of a method in the pre-state:

$$T_{\mathsf{m}}(\ell,\sigma,v,\sigma') \equiv \{P\}\sigma.\ell.\mathsf{n}\{Q\} \Rightarrow T(\ell,\sigma,v,\sigma'),$$

where T is a transition specification. If method update is possible one could even want to specify a re-definition of a method (say n):

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

Fortunately, one can get rid of condition (iii) for this kind of specifications.

Corollary 4.7. *If a predicate* $T_{\mathsf{m}} \subseteq \mathsf{Loc}\times\mathsf{St}\times\mathsf{Val}\times\mathsf{St}$ *is of the form*

$$T_{\mathsf{m}}(\ell,\sigma,v,\sigma') \equiv \{P\}\sigma.\ell.\mathsf{n}\,\{Q\} \quad\Rightarrow\quad \{P'\}\,\sigma'.\ell.\mathsf{n}\,\{Q'\} \wedge T(\ell,\sigma,v,\sigma')$$

such that Q is downward-closed, Q' is Scott-closed, and T fulfils (iii) *of Theorem* 4.4 (*which it does, for example, if it only refers to the fields stored in σ and σ'*) *then also T_{m} fulfils condition* (iii).

Proof. The predicate T_{m} is Scott-closed in the fourth argument by Lemma 4.6 (as Q' is Scott-closed by definition) and condition (iii) for T. Monotonicity in the second argument follows again from condition (iii) for T, and by the fact that $\{P\}(-)\{Q\}$ is downward-closed if Q is. □

In order to be able to specify methods as parameters (in the sense of higher-order functions) this is still not sufficient. As in Specification Logic [22] (see also [3]), one needs quantification over (arbitrary) method specifications. This can be done by quantifying over P and Q in an transition specification.

Corollary 4.8. *If a predicate* $T_{\mathsf{m}} \subseteq \mathsf{Loc}\times\mathsf{St}\times\mathsf{Val}\times\mathsf{St}$ *is of the form*

$$\begin{aligned} T_{\mathsf{m}}(\ell,\sigma,v,\sigma') = \forall P,Q \in \mathscr{L}_{\mathsf{Loc}}(\mathsf{St}).\ & \{P\}\sigma.\ell.\mathsf{n}\{Q\} \Rightarrow \\ & \{P'[P,Q]\}\sigma'.\ell.\mathsf{n}\{Q'[P,Q]\} \wedge T(\ell,\sigma,v,\sigma') \end{aligned}$$

such that $Q'[P,Q]$ is a Scott-closed predicate (*that may use P and Q*) *then if T fulfils condition* (iii) *of Theorem* 4.4, *so does T_{m}*.

Proof. The proof is like above since all quantified predicates are Scott-closed and admissible predicates are closed under universal quantification. □

Example 4.9. Consider a simple listener-notify-mechanism [23]. A callback object, *cb*, contains a listener object (usually a vector of such objects), a notify method, and some local state, say a field f. The listener object is unknown, as it is updated (by field update) on the fly, but it is known to have a method run which is called upon notification.

The specification of the notify method of *cb* may look as follows:

$$\begin{aligned} T_{\mathsf{notify}}(\ell,\sigma,v,\sigma') \equiv\ & \mathsf{listener} \in \mathsf{dom}\,\sigma.\ell\wedge \\ & (\sigma.\ell.\mathsf{listener}) \in \mathsf{dom}\,\sigma\wedge \\ & \mathsf{run} \in \sigma.(\sigma.\ell.\mathsf{listener})\wedge \\ & \forall P,Q \in \mathscr{L}_{\mathsf{Loc}}(\mathsf{St}). \\ & \{P\}\sigma.(\sigma.\ell.\mathsf{listener}).\mathsf{run}\{Q\} \wedge P(\sigma.\ell.\mathsf{listener},\sigma) \Rightarrow Q(v,\sigma'). \end{aligned}$$

The first three lines just ensure the presence of the right methods including the existence of the object $\mathsf{listener}$ in the store. The last two lines ensure that notify behaves like the run method of the $\mathsf{listener}$ object. This reflects the fact that the run method will be called upon notification. By using the quantification over P and Q this specification works for an arbitrary $\mathsf{listener}$ and its run method.

For example, $\langle\ell,\sigma\rangle$ fulfils the specification above if

$$\sigma.\ell = \{\!|\mathsf{listener} = \ldots, \mathsf{notify} = \lambda(l,\sigma').\ \sigma'.(\sigma'.l.\mathsf{listener}).\mathsf{run}(\sigma'.l.\mathsf{listener},\sigma')|\!\}$$

4.3. Non-existing specifications

Before showing that particular object specifications do not exist we prove the following auxiliary lemma that deals with transition specifications that may also refer to the non flat part of the store, i.e. to some method closures.

Lemma 4.10. *Let* $A \subseteq \mathsf{Loc}\times\mathsf{St}\times[\mathsf{Loc}\times\mathsf{St}\rightharpoonup\mathsf{Val}\times\mathsf{St}]$ *and* $S\subseteq\mathsf{Loc}\times\mathsf{St}$ with
(0) $\forall\ell,\sigma.\ \langle\ell,\sigma\rangle\in S\Leftrightarrow[\forall(\ell',\sigma')\in S.\ A(\ell',\sigma',\sigma.\ell.\mathsf{m})]$.
If ℓ *is a location and* σ *a state satisfying*
(1) $\forall\ell',\sigma'.\ A(\ell',\sigma',\sigma'.\ell'.\mathsf{m})\Rightarrow A(\ell',\sigma',\sigma.\ell.\mathsf{m})$
then
(2) $A(\ell,\sigma,\sigma.\ell.\mathsf{m})$.

Proof. From (0) it follows that

$$(\dagger)\ \forall\ell',\sigma'.\ \langle\ell',\sigma'\rangle \in S \Rightarrow A(\ell',\sigma',\sigma'.\ell'.\mathsf{m})$$

and, therefore, by (1) that

$$\forall\ell',\sigma'.\langle\ell',\sigma'\rangle \in S \Rightarrow A(\ell',\sigma',\sigma.\ell.\mathsf{m}),$$

i.e. that $(\ell,\sigma)\in S$. Thus, by (†) we have $A(\ell,\sigma,\sigma.\ell.\mathsf{m})$. □

To give an example of a non-existing object specification, we will exhibit a transition specification T_{m}, a location ℓ, and a store σ such that there does not exist $S\subseteq\mathsf{Loc}\times\mathsf{St}$ satisfying

$$\langle\ell,\sigma\rangle \in S \Leftrightarrow \forall\langle\ell',\sigma'\rangle \in S.\ \sigma.\ell.\mathsf{m}(\ell',\sigma')\!\downarrow\,\Rightarrow T_{\mathsf{m}}(\ell',\sigma',\sigma.\ell.\mathsf{m}(\ell',\sigma')).$$

For such a specification the restrictive assumption of Theorem 4.4—that T_{m} must only refer to the flat part of the store—must necessarily be violated, but also condition (iii) from Theorem 4.4 cannot hold.

Example 4.11. Consider the following object specification:

$$\langle \ell, \sigma \rangle \in S \equiv \forall \langle \ell', \sigma' \rangle \in S.\ \sigma.\ell.\mathsf{m}(\ell', \sigma')\downarrow \Rightarrow \sigma'.\ell'.\mathsf{f}\downarrow \Rightarrow T(\ell', \sigma', \sigma.\ell.\mathsf{m}(\ell', \sigma')),$$

where

$$T(\ell', \sigma', v, \sigma'') \equiv \exists n \in \mathbb{N}.\ \sigma'.\ell'.\mathsf{m}(\ell', \sigma'[\ell'.\mathsf{f} := n])\uparrow.$$

Note that T is not monotonic in σ', its second argument. Let $A(\ell, \sigma, h)$ denote the property

$$h(\ell, \sigma)\downarrow \Rightarrow \sigma.\ell.\mathsf{f}\downarrow \Rightarrow \exists n \in \mathbb{N}.\ h(\ell, \sigma[\ell.\mathsf{f} := n])\uparrow$$

then the specification S above can be reformulated as

$$\langle \ell, \sigma \rangle \in S \equiv \forall \langle \ell', \sigma' \rangle \in S.\ A(\ell', \sigma', \sigma.\ell.\mathsf{m})$$

i.e. condition (0) of Lemma 4.10 holds for A. Now for (ℓ, σ) with $\sigma.\ell.\mathsf{f} = 0$ and

$$\sigma.\ell.\mathsf{m} = [\![\varsigma(x)\ \mathtt{if}\ x.\mathsf{f} = 0\ \mathtt{then}\ x\ \mathtt{else}\ x.\mathsf{f} := x.\mathsf{f} - 1; x.\mathsf{m}()]\!]$$

one easily verifies that (1) holds but (2) is false, contradicting Lemma 4.10(2).

More natural counterexamples are expected by semantic modelling of logics for object calculi using Hoare triples like the one suggested in [3].

5. Applications

The semantics of specifications can be employed to verify and analyse programming logics given syntactically by proof rules.

5.1. Soundness of the Abadi and Leino logic denotationally

Our notion of specification suggests that the logic proposed in [2] is correct. It does that in a very intuitive way as every specifications has a unique denotation.

Claim 5.1. *The object creation rule of the Abadi and Leino logic is correct w.r.t. our semantics.*

Proof. An object specification S in [2] reads as follows:

$$S = [\mathsf{f}_i : A_i^{i \in 1,\dots,n}, \mathsf{m}_j : \varsigma(y)B_j :: T_j^{j \in 1,\dots,m}],$$

where, again, the A_i are field specifications. The predicates B_j are the result specifications for methods m_j. They could be basic types or compound object specifications. The predicates T_j are the transition specifications for m_j. If we interpret all these predicates denotationally it follows from Theorem 4.4 that they give rise to a unique predicate that is as in Definition 3.3 and serves as the denotational interpretation of S.

In the logic of *Adabi and Leino* [2] the judgment

$$E \vdash t : B :: T$$

means that in context E the result of program term t fulfils result specification B and its behaviour satisfies transition specification T.

The object formation rule roughly looks as follows (where "…" denote omitted parts which are not relevant to our case).

$$\frac{E \vdash x_i : A_i :: \ldots^{i\in 1..n} \quad E, y : S \vdash b_j : B_j :: T_j^{j\in 1..m}}{E \vdash [\mathsf{f}_i = x_i^{i\in 1..n}, \mathsf{m}_j = \varsigma(y)b_j^{j\in 1..m}] : S :: \ldots}.$$

If we ignore the context E this rule matches Definition 3.3. The quantification over arbitrary pairs $\langle \ell', \sigma' \rangle \in S$ is forced by this rule due to the assumption $y : S$ in the premise $E, y : S \vdash b_j : B_j :: T_j$.

Since in the Abadi and Leino logic the transition specifications T_j can only refer to the *flat part of the store* existence of S can be guaranteed.

A full soundness proof of the Abadi and Leino logic is omitted due to space limitation but a corresponding paper is in preparation. Invariance of specifications of contexts and sub-specifications contribute to the difficulties of extending the ideas presented here. It is expected that such a "denotational" soundness proof conveys more intuition than the one presented in [2]. □

5.2. *Possible extensions of the Abadi and Leino logic*

Dealing with object specifications denotationally does not only yield a concise explanation of the Abadi and Leino logic but also suggests extensions.

5.2.1. *Invariants*

By contrast to Abadi and Leino the predicates A and B_{m} may contain recursive occurrences of the specification itself. If a field is required to fulfill the same specification as the ambient object then one needs recursion in A. If a result of method m is required to contain (or be itself) an object fulfilling the same specification as the ambient object, one needs recursion in B_{m}. In [12] a variation is presented that allows for recursive object descriptions but requires that methods are declared in advance.

The approach presented here allows for even more. When defining a specification we can express the fact that the original object (callee) still fulfils the specification after any of the methods has been called. Such specifications can be described as follows:

$$\begin{aligned}\langle \ell, \sigma \rangle \in S \equiv A(S)(\ell, \sigma) \wedge &\\ &\forall \mathsf{m} \in \mathscr{M}.\ \forall \ell' \in \mathsf{Loc}.\ \forall \sigma' \in \mathsf{St}.\ \langle \ell', \sigma' \rangle \in S \Rightarrow\\ &\quad \forall v \in \mathsf{Val}.\ \forall \sigma'' \in \mathsf{St}.\ \sigma.\ell.\mathsf{m}(\ell', \sigma') = \langle v, \sigma'' \rangle \Rightarrow\\ &\qquad B_{\mathsf{m}}(S)(v, \sigma'') \wedge T_{\mathsf{m}}(\ell', \sigma', v, \sigma'') \wedge (\ell', \sigma'') \in S.\end{aligned}$$

Notice how the last part of the conjunction $(\ell', \sigma'') \in S$ establishes S as invariant. Such specifications be used as *invariants* in specifications of software packages (classes).

They cannot be dealt with in [2] and the denotational approach advertised here may help to understand if (and how) their calculus may be extended to deal with such invariants.

Existence of invariant specifications can be shown analogously as in Theorem 4.4.

5.2.2. Method update

If method updates are allowed then Corollary 4.8 suggests a way to specify such method updates using Hoare-triples for method closures in the store. Existence of those (recursive) specifications is still guaranteed by the corollaries of Section 4.2.2. An open question is how a sound and sufficiently useful proof calculus can be devised that allows for method updates. The semantic approach may be helpful here. Invariants, though, will be harder to establish because they do not fit into the pattern of the original definition (cf. Definition 3.3).

6. Conclusions

We have shown that a denotational approach to programming logics for object calculi leads to a better understanding of the implicit recursion of object specifications and their reasoning principles. Since the notion of specification encodes the object introduction rule of the logic, the soundness of this rule is equivalent to the existence of the specification. To guarantee existence one has to be careful with reasoning on the non-flat method part of the store.

It should be possible to deal with other, similar, object calculi and logics in the same *denotational* way. The analysis of further languages should be fruitful in the quest for more (natural) counterexamples.

A comparison with class-based languages has been attempted in [19] but using a closed world assumption, where classes cannot be added compositionally. In [18] a modular simple class based semantics and modular verification rules have been discussed. Though dynamic loading of classes at runtime can only be done using the more sophisticated techniques presented in this paper.

Other issues to be tackled include a complete soundness proof for the Abadi and Leino logic possibly extended by invariants and reasoning principles for methods in the store. Additional language features like garbage collection or method parameters should be investigated. The development of a logic over a *typed* semantics of the object calculus (with subtyping including method parameters) is challenging too.

Recursive methods can be programmed in the object calculus without explicit recursion due to the recursive higher-order definition of the underlying store. This is a particular instance of "*recursion through the store*" a more general variant of which allows unrestricted execution of code stored in memory. Such a rather liberal usage of higher-order store needs to be modelled by simpler (but similar) domain equations.

In this paper we did not commit ourselves to any particular object logic not to classical or intuitionistic logic. Spatial or separation logic [4,13] is a prospective candidate for such an object logic as it simplifies handling of aliases addressing the heap. It remains to be seen whether predicates in such a logic pose any problems to the presented approach.

Acknowledgements

Thanks to Cristiano Calcagno and Peter O'Hearn for discussions on "recursion through the store" and other related matters. Thanks to Jan Schwinghammer for proof reading.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, Berlin, 1996.

[2] M. Abadi, K. Leino, A logic of object-oriented programs, in: M. Bidoit, M. Dauchet (Eds.), Theory and Practice of Software Development: Proc./TAPSOFT '97, 7th Internat. Joint Conf. CAAP/FASE, Lecture Notes in Computer Science, Vol. 1214, Springer, Berlin, 1997, pp. 682–696.

[3] C. Calcagno, P. O'Hearn, A logic for objects, talk given in March 2001.

[4] C. Calcagno, P.W. O'Hearn, On garbage and program logic, in: FoSSaCS, Lecture Notes in Computer Science, Vol. 2030, Springer, Berlin, 2001, pp. 137–151.

[5] W. Cook, A denotational semantics of inheritance, Ph.D. Thesis, Tech. Rep. CS-89-33, Department of Computer Science, Brown University, 1989.

[6] F. de Boer, A WP-calculus for OO, in: W. Thomas (Ed.), Foundations of Software Science and Computations Structures, Lecture Notes in Computer Science, Vol. 1578, Springer, Berlin, 1999.

[7] P. Freyd, Algebraically complete categories, in: A. Carboni, M. Pedicchio, G. Rosolini (Eds.), Proc. 1990 Como Category Theory Conf, Lecture Notes in Mathematics, Vol. 1488, Springer, Berlin, 1991, pp. 95–104.

[8] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, H. Tews, Reasoning about Java classes, in: Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), ACM SIGPLAN Notices 33 (10) (1998) 329–340.

[9] S. Kamin, Inheritance in SMALLTALK-80: a denotational definition, in: Principles of Programming Languages ACM Press, New York, 1988, pp. 80–87.

[10] S. Kamin, U. Reddy, Two semantic models of object-oriented languages, in: C.A. Gunter, J.C. Mitchell (Eds.), Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design, The MIT Press, Cambridge, MA, 1994, pp. 464–495.

[11] G. Leavens, Modular specification and verification of object-oriented programs, IEEE Software 8 (4) (1991) 72–80.

[12] K.R.M. Leino, Recursive object types in a logic of object-oriented programs, Nordic J. Comput. 5 (4) (1998) 330–360.

[13] P.W. O'Hearn, J.C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: CSL, Lecture Notes in Computer Science, Vol. 2142, Springer, Berlin, 2001, pp. 1–19.

[14] A.M. Pitts, Relational properties of domains, Inform. and Comput. 127 (1996) 66–90 (A preliminary version of this work appeared as Cambridge University Computer Laboratory Tech. Rept. No. 321, December 1993.).

[15] A. Poetzsch-Heffter, P. Müller, Logical foundations for typed object-oriented languages, in: D. Gries, W. de Roever (Eds.), Programming Concepts and Methods, Elsevier Amsterdam, 1998.

[16] U. Reddy, Objects and classes in Algol-like languages, Inform. and Comput. 172 (2002) 63–97.

[17] B. Reus, A logic of recursive objects (abstract), in: A.M.D. Moreira, S. Demeyer (Eds.), Object-oriented Technology, ECOOP '99 Workshop Reader, Lecture Notes in Computer Science, Vol. 1743, Springer, Berlin, 1999, p. 107.

[18] B. Reus, Class based vs. object based: a denotational comparison, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology. Lecture Notes in Computer Science, Vol. 2422, Springer, Berlin, 2002, pp. 473–488.

[19] B. Reus, Modular semantics and logics of classes, in: M. Baaz, J.A. Makowsky (Eds.), Computer Science Logic. Lecture Notes in Computer Science, Vol. 2803, Springer, Berlin, 2003, pp. 456–469.
[20] B. Reus, T. Streicher, Semantics and logics of objects, in: Proc. 17th Symp. Logic in Computer Science, 2002, pp. 113–122.
[21] B. Reus, M. Wirsing, R. Hennicker, A Hoare-calculus for verifying Java realizations of OCL-constrained design models, in: FASE 2001. Lecture Notes in Computer Science, Vol. 2029, Springer, Berlin, 2001, pp. 300–317.
[22] J. Reynolds, An introduction to specification logic, in: E.M. Clarke, D. Kozen (Eds.), Logic of Programs, Lecture Notes in Computer Science, Vol. 164, Springer, Berlin, 1984, p. 442.
[23] C. Szyperski, Component Software: Beyond Object-oriented Programming, 2nd edition, Component Software Series, Addison-Wesley, Reading, MA, 2002.