Denotational Semantics for Abadi and Leino's Logic of Objects

Bernhard Reus and Jan Schwinghammer*

Informatics, University of Sussex, Brighton, UK
Fax +44 1273 877873
{bernhard, j.schwinghammer}@sussex.ac.uk

Abstract. Abadi-Leino Logic is a Hoare-calculus style logic for a simple imperative and object-based language where every object comes with its own method suite. Consequently, methods need to reside in the store ("higher-order store"). We present a new soundness proof for this logic using a denotational semantics where object specifications are recursive predicates on the domain of objects. Our semantics reveals which of the limitations of Abadi and Leino's logic are deliberate design decisions and which follow from the use of higher-order store. We discuss the implications for the development of other, more expressive, program logics.

1 Introduction

When Hoare presented his seminal work about an *axiomatic basis of computer programming* [7], high-level languages had just started to gain broader acceptance. Meanwhile programming languages are evolving ever more rapidly, whereas verification techniques seem to be struggling to keep up. For object-oriented languages several formal systems have been proposed, e.g. [2, 6, 14, 13, 5, 21, 20]. A "standard" comparable to the Hoare-calculus for imperative While-languages [4] has not yet emerged. Nearly all the approaches listed above are designed for class-based languages (usually a sub-language of sequential Java), where method code is known statically.

One notable exception is Abadi and Leino's work [2] where a logic for an object-based language is introduced that is derived from the imperative object calculus with first-order types, **imp**ς, [1]. In object-based languages, every object contains its own suite of methods. Operationally speaking, the store for such a language contains code (and is thus called *higher-order store*) and modularity is for free simply by the fact that all programs can depend on the objects' code in the store. We therefore consider object-based languages ideal for studying modularity issues that occur also in class-based languages. Class-based programs can be compiled into object-based ones (see [1]), and object-based languages can

* Supported by the EPSRC under grant GR/R65190/01, "Programming Logics for Denotations of Recursive Objects".

naturally deal with classes defined on-the-fly, like inner classes and classes loaded at run-time (cf. [16, 17]).

Abadi and Leino's logic is a Hoare-style system, dealing with partial correctness of object expressions. Their idea was to enrich object types by method specifications, also called *transition relations*, relating pre- and post-execution states of program statements, and *result specifications* describing the result in case of program termination. Informally, an object satisfies such a specification

$$A \equiv [\mathsf{f}_i \colon A_i{}^{i=1\ldots n},\ \mathsf{m}_j \colon \varsigma(y_j)B_j \colon\colon T_j{}^{j=1\ldots m}]$$

if it has fields f_i satisfying A_i and methods m_j that satisfy the transition relation T_j and, in case of termination of the method invocation, their result satisfies B_j. However, just as a method can use the *self*-parameter, we can assume that an object a itself satisfies A in both B_j and T_j when establishing that A holds for a. This yields a powerful and convenient proof principle for objects.

We are going to present a new proof using a (untyped) denotational semantics (of the language and the logic) to define validity. Every program and every specification have a meaning, a *denotation*. Those of specifications are simply predicates on (the domain of) objects. The properties of these predicates provide a description of inherent limitations of the logic. Such an approach is not new, it has been used e.g. in LCF, a logic for functional programs [11].

The difficulty in this case is to establish predicates that provide the powerful reasoning principle for objects. Reus and Streicher have outlined in [19] how to use some classic domain theory [12] to guarantee existence and uniqueness of appropriate predicates on (isolated) objects. In an object-calculus program, however, an object may depend on other objects (and its methods) in the store. So object specifications must depend on specifications of other objects in the store which gives rise to "store specifications" (already present in the work of Abadi and Leino).

For the reasons given above, this paper is not "just" an application of the ideas in [19]. Much care is needed to establish the important invariance property of Abadi-Leino logic, namely that proved programs preserve store specifications. Our main achievement, in a nutshell, is that we have successfully applied the ideas of [19] to the logic of [2] to obtain a soundness proof that can be used to *analyse this logic* and to *develop similar but more powerful program logics* as well.

Our soundness proof is not just "yet another proof" either. We consider it complementary (if not superior) to the one in [2] which relies on the operational semantics of the object calculus and does not assign proper "meaning" to specifications. Our claim is backed up by the following reasons:

- By using denotational semantics we can introduce a clear notion of validity with no reference to derivability. This helps clarifying *what the proof is actually stating* in the first place.
- We can extend the logic easily e.g. for recursive specifications. This has been done for the Abadi-Leino logic in [9] but for a slightly different language with nominal subtyping.

– Some essential and unavoidable restrictions of the logic are revealed and justified.
– Analogously, it is revealed where restrictions have been made for the sake of simplicity that could be lifted to obtain a more powerful logic. For example, in [2] transition specifications cannot talk about methods at all.
– Our proof widens the audience for Abadi and Leino's work to semanticists and domain theorists.

The outline of this paper is as follows. In the next section, syntax and semantics of the object-calculus are presented. Section 3 introduces the Abadi-Leino logic and the denotational semantics of its object specifications. A discussion about store specifications and their semantics follows (Section 4). The main result is in Section 5 where the logic is proved sound. Of the various extensions discussed in Section 6, we present recursive specifications in some detail (Section 6.2). Section 7 concludes with a brief comparison to the original proof [2].

When presenting the language and logic, we deliberately keep close to the original presentation [2]. For a full version of this paper containing detailed proofs we refer to the technical report [18].

2 The Object Calculus

Below, we review the language of [2], which is based on the imperative object calculus of Abadi and Cardelli [1]. Following [19] we give a denotational semantics. The syntax of terms is defined by

$$\begin{aligned} a, b ::= {} & x \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ x\ \texttt{then}\ a\ \texttt{else}\ b \mid \texttt{let}\ x = a\ \texttt{in}\ b \\ \mid {} & [\mathsf{f}_i = x_i{}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j)b_j{}^{j=1\ldots m}] \mid x.\mathsf{f} \mid x.\mathsf{f} := y \mid x.\mathsf{m} \end{aligned}$$

where $\mathsf{f} \in \mathcal{F}$ and $\mathsf{m} \in \mathcal{M}$ range over countably infinite sets of *field* and *method names*, resp. Object construction $[\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j]$ allocates new storage and returns (a reference to) an object containing fields f_i (with initial value the value of x_i) and methods m_j. In a method m_j, ς is a binder for the *self* parameter y_j in the method body b_j. During method invocation, the method body is evaluated with the self parameter bound to the host object.

We extend the syntax with integer constants and operations, and consider an object-based modelling of a bank account as an example:

```
acc(x) ≡ [balance = 0,
          deposit10 = ς(y) let z = y.balance+10 in y.balance:=z,
          interest = ς(y) let r = x.manager.rate in
                               let z = y.balance∗r/100 in y.balance:=z]
```

Note how the self parameter y is used in both methods to access the `balance` field. Object *acc* depends on a "managing" object x in the context that provides the interest rate, through a field `manager`, for the `interest` method.

Semantics of Objects. We work in the category PreDom of predomains (cpos that do not necessarily contain a least element) and partial continuous functions. Let $A \rightharpoonup B$ denote the partial continuous function space between predomains A and B. For $f \in A \rightharpoonup B$ and $a \in A$ we write $f(a)\downarrow$ if f applied to a is defined, and $f(a)\uparrow$ otherwise.

If L is a set, then $\mathcal{P}(L)$ is its powerset, $\mathcal{P}_{\mathsf{fin}}(L)$ denotes the set of its finite subsets, and A^L is the set of all total functions from L to A. For a countable set $\mathbb{L}$ and a predomain A we write $\mathsf{Rec}_{\mathbb{L}}(A) = \sum_{L \in \mathcal{P}_{\mathsf{fin}}(\mathbb{L})} A^L$ for the predomain of *records* with entries from A and labels from $\mathbb{L}$. Note that $\mathsf{Rec}_{\mathbb{L}}$ extends to a locally continuous endofunctor on PreDom.

We write $\{\!|l_1 = a_1, \ldots, l_n = a_n|\!\}$ for a record $r = (L, f \in A^L)$, with labels $L = \{l_1, \ldots, l_n\}$ and corresponding entries $f(l_i) = a_i$. Update (and extension) $r[l := a]$ is defined in the obvious way. Selection of labels is written $r.l$.

The language of the previous section finds its interpretation within the following system of recursively defined predomains in PreDom:

$$\begin{aligned} \mathsf{Val} &= \mathsf{BVal} + \mathsf{Loc} & \mathsf{Ob} &= \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \\ \mathsf{St} &= \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}) & \mathsf{Cl} &= \mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St} \end{aligned} \tag{1}$$

where Loc is a countably infinite set of *locations* ranged over by l, and BVal is the of truth values *true* and *false*, considered as flat predomains.

Let $\mathsf{Env} = \mathsf{Var} \rightarrow_{\mathsf{fin}} \mathsf{Val}$ be the set of *environments*, i.e. maps between Var and Val with finite domain. Given an environment $\rho \in \mathsf{Env}$, the interpretation $[\![a]\!]\rho$ of an object expression a in $\mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St}$ is given in Table 1, where the (strict) semantic **let** is also "strict" wrt. error. Note that for $o \in \mathsf{Ob}$ we just write $o.\mathsf{f}$ and $o.\mathsf{m}$ instead of $\pi_1(o).\mathsf{f}$ and $\pi_2(o).\mathsf{m}$, respectively. Similarly, we omit the injections for elements of $\mathsf{Val} + \{\mathsf{error}\}$. Because Loc is assumed to be infinite, the condition $l \notin \mathsf{dom}(\sigma)$ in the case for object creation can always be satisfied, i.e., object creation will never raise error because we run out of memory.

We will also use a projection to the part of the store that contains just data in Val (no closures), $\pi_{\mathrm{Val}} : \mathsf{St} \rightarrow \mathsf{St}_{\mathsf{Val}}$ defined by $(\pi_{\mathrm{Val}}\,\sigma).l.\mathsf{f} = \sigma.l.\mathsf{f}$, where $\mathsf{St}_{\mathsf{Val}} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}))$. We refer to $\pi_{\mathrm{Val}}(\sigma)$ as the *flat part* of σ.

3 Abadi-Leino Logic

We briefly recall Abadi and Leino's logic. For a more detailed presentation see [2, 8] or the technical report [18]. A *transition relation* T is a first-order formula over program variables that relates pre- and post-execution states of computations. There are function symbols $\grave{\sigma}$, $\acute{\sigma}$ and result that refer to the (flat parts of the) initial and final stores, and the result of a computation, resp. For instance, $\grave{\sigma}(x, \mathsf{f})$ denotes the value of $x.\mathsf{f}$ in the initial store, and analogously for $\acute{\sigma}$. Predicate symbols include T_{res}, T_{upd} and T_{obj} with the following meaning:

- $T_{\mathsf{res}}(x)$ holds if $\mathsf{result} = x$, and the (flat part of the) store remains unchanged
- $T_{\mathsf{upd}}(x, \mathsf{f}, y)$ holds if $\mathsf{result} = x$, $\acute{\sigma}(x, \mathsf{f}) = y$, and $\grave{\sigma}$ equals $\acute{\sigma}$ everywhere else
- $T_{\mathsf{obj}}(\mathsf{f}_1 = x_1, \ldots, \mathsf{f}_n = x_n)$ holds if result denotes a fresh location such that $x_i = \acute{\sigma}(\mathsf{result}, \mathsf{f}_i)$ for all i, and the store remains unchanged otherwise.

Table 1. Denotational semantics

$$
\begin{array}{lll}
\llbracket x \rrbracket \rho\sigma & = & \begin{cases} (\rho(x), \sigma) & \text{if } x \in \mathsf{dom}(\rho) \\ (\mathsf{error}, \sigma) & \text{otherwise} \end{cases} \\
\llbracket \mathtt{true} \rrbracket \rho\sigma & = & (\mathit{true}, \sigma) \\
\llbracket \mathtt{false} \rrbracket \rho\sigma & = & (\mathit{false}, \sigma) \\
\llbracket \mathtt{if}\ x\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2 \rrbracket \rho\sigma & = & \begin{cases} \llbracket b_1 \rrbracket \rho\sigma' & \text{if } \llbracket x \rrbracket \rho\sigma = (\mathit{true}, \sigma') \\ \llbracket b_2 \rrbracket \rho\sigma' & \text{if } \llbracket x \rrbracket \rho\sigma = (\mathit{false}, \sigma') \\ (\mathsf{error}, \sigma') & \text{if } \llbracket x \rrbracket \rho\sigma = (v, \sigma') \text{ for } v \notin \mathsf{BVal} \end{cases} \\
\llbracket \mathtt{let}\ x\ \mathtt{=}\ a\ \mathtt{in}\ b \rrbracket \rho\sigma & = & \mathbf{let}\ (v, \sigma') = \llbracket a \rrbracket \rho\sigma\ \mathbf{in}\ \llbracket b \rrbracket \rho[x := v]\sigma' \\
\llbracket [\mathsf{f}_i = x_i{}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j)b_j{}^{j=1\ldots m}] \rrbracket \rho\sigma & = & \begin{cases} (l, \sigma[l := (o_1, o_2)]) & \text{if } x_i {\in} \mathsf{dom}(\rho), 1 \le i \le n \\ (\mathsf{error}, \sigma) & \text{otherwise} \end{cases} \\
 & & \text{where } l \notin \mathsf{dom}(\sigma) \\
 & & \qquad o_1 = \{\!| \mathsf{f}_i = \rho(x_i) |\!\}^{i=1\ldots n} \\
 & & \qquad o_2 = \{\!| \mathsf{m}_j = \lambda\sigma. \llbracket b_j \rrbracket \rho[y_j := l]\sigma |\!\}^{j=1\ldots m} \\
\llbracket x.\mathsf{f} \rrbracket \rho\sigma & = & \mathbf{let}\ (l, \sigma') = \llbracket x \rrbracket \rho\sigma \\
 & & \mathbf{in} \begin{cases} (\sigma'.l.\mathsf{f}, \sigma') & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{f} \in \mathsf{dom}(\sigma'.l) \\ (\mathsf{error}, \sigma') & \text{otherwise} \end{cases} \\
\llbracket x.\mathsf{f} := y \rrbracket \rho\sigma & = & \mathbf{let}\ (l, \sigma') = \llbracket x \rrbracket \rho\sigma, (v, \sigma'') = \llbracket y \rrbracket \rho\sigma' \\
 & & \mathbf{in} \begin{cases} (l, \sigma''[l := \sigma''.l[\mathsf{f} := v]]) & \text{if } l \in \mathsf{dom}(\sigma'') \\ & \text{and } \mathsf{f} \in \mathsf{dom}(\sigma''.l) \\ (\mathsf{error}, \sigma'') & \text{otherwise} \end{cases} \\
\llbracket x.\mathsf{m} \rrbracket \rho\sigma & = & \mathbf{let}\ (l, \sigma') = \llbracket x \rrbracket \rho\sigma \\
 & & \mathbf{in} \begin{cases} \sigma'.l.\mathsf{m}(\sigma') & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{m} \in \mathsf{dom}(\sigma'.l) \\ (\mathsf{error}, \sigma') & \text{otherwise} \end{cases}
\end{array}
$$

Specifications combine transition relations for each method as well as the specifications of their results into a single specification for the whole object. They generalise the first-order types of [1], and are

$$A, B ::= \mathit{Bool} \mid [\mathsf{f}_i{:}\, A_i{}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}]$$

where each T_j is a transition relation, and in general both B_j and T_j depend on the self parameter y_j.

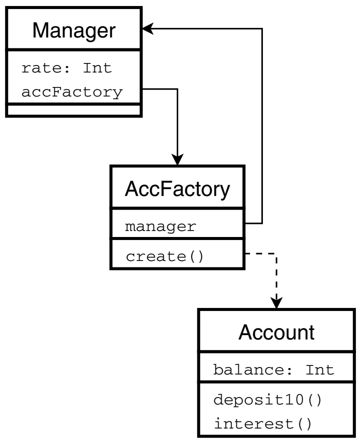
Table 2 shows a specification for bank accounts as in the previous example.[1] Observe how the specification $T_{\mathtt{interest}}$ depends not only on the self parameter y of the host object but also on the statically enclosing object x.

Judgments of the logic are of the form $x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A :: T$. Informally, such a judgment means that if program a terminates when executed in a context where program variables $x_1, \ldots, x_n$ denote values that satisfy specifications $A_1, \ldots, A_n$, resp., then A describes properties of the result, and T describes the dynamic behaviour of a.

[1] Note that although we are using UML-like notation, these diagrams actually stand for individual objects, not classes – in fact there are no classes in the language.

Table 2. An example of transition and result specifications

$$\begin{aligned}
T_{\texttt{deposit}}(y) &\equiv \exists z.z = \grave{\sigma}(y, \texttt{balance}) \\
&\quad \wedge T_{\textsf{upd}}(y, \texttt{balance}, z + 10) \\
T_{\texttt{interest}}(x, y) &\equiv \exists z.z = \grave{\sigma}(y, \texttt{balance}) \\
&\quad \wedge \exists m.m = \grave{\sigma}(x, \texttt{manager}) \\
&\quad \wedge \exists r.r = \grave{\sigma}(m, \texttt{rate}) \\
&\quad \wedge T_{\textsf{upd}}(y, \texttt{balance}, z * r/100) \\
T_{\texttt{create}}(x) &\equiv T_{\textsf{obj}}(\texttt{balance} = 0) \\
A_{\texttt{Account}}(x) &\equiv [\texttt{balance} : \mathit{Int}, \\
&\quad \texttt{deposit10} : \varsigma(y)[] :: T_{\texttt{deposit}}(y), \\
&\quad \texttt{interest} : \varsigma(y)[] :: T_{\texttt{interest}}(x, y)] \\
A_{\texttt{AccFactory}} &\equiv [\texttt{manager} : [\texttt{rate} : \mathit{Int}], \\
&\quad \texttt{create} : \varsigma(x) A_{\texttt{Account}}(x) :: T_{\texttt{create}}(x)] \\
A_{\texttt{Manager}} &\equiv [\texttt{rate} : \mathit{Int}, \\
&\quad \texttt{accFactory} : A_{\texttt{AccFactory}}]
\end{aligned}$$



We can use the proof rules of Abadi and Leino's logic to derive the judgment

$$x{:}A_{\texttt{AccFactory}} \vdash \mathit{acc}(x) : A_{\texttt{Account}}(x) :: T_{\textsf{obj}}(\texttt{balance} = 0) \tag{2}$$

for the *acc* object. In the logic there is one rule for each syntactic form of the language. As indicated in the introduction, the most interesting and powerful rule of the logic is the object introduction rule,

$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\, {A_i}^{i=1\ldots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}] \\ \Gamma \vdash x_i{:}A_i{::}T_{\textsf{res}}(x_i)^{i=1\ldots n} \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j^{j=1\ldots m} \end{array}}{\Gamma \vdash [\mathsf{f}_i = {x_i}^{i=1\ldots n}, \mathsf{m}_j = \varsigma(y_j){b_j}^{j=1\ldots m}] : A :: T_{\textsf{obj}}(\mathsf{f}_i = {x_i}^{i=1\ldots n})}$$

In order to establish that the newly created object satisfies specification A one has to verify the methods b_j. When doing that one can *assume* that the host object (through the self parameter y_j) already satisfies A. Essentially, this causes the semantics of store specifications, introduced in the next section, to be defined by a mixed-variant recursion.

Using the object introduction rule, (2) can be reduced to a trivial proof obligation for the field `balance`, a judgment for the method `deposit10`,

$$\Gamma \vdash \texttt{let } z\texttt{=(}y\texttt{.balance)+10 in } y\texttt{.balance:=}z : [] :: T_{\texttt{deposit}}(y) \tag{3}$$

where Γ is the context $x{:}A_{\texttt{AccFactory}}, y{:}A_{\texttt{Account}}(x)$, and a similar judgment for the method `interest`. A proof of (3) involves showing

$$\Gamma \vdash (y\texttt{.balance)+10} : \mathit{Int} :: T_{\textsf{res}}(\grave{\sigma}(y, \texttt{balance}) + 10) \tag{4}$$

$$\Gamma, z{:}\mathit{Int} \vdash y\texttt{.balance:=}z : [] :: T_{\textsf{upd}}(y, \texttt{balance}, z) \tag{5}$$

for the constituents of the let expression. These can be proved from the rules for field selection and field update, resp., which have the general form

$$\frac{\mathit{\Gamma} \vdash x{:}[\mathsf{f}{:}A]{::}T_{\mathsf{res}}(x)}{\mathit{\Gamma} \vdash x.\mathsf{f}{:}A{::}T_{\mathsf{res}}(\grave{\sigma}(x,\mathsf{f}))} \qquad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\,A_i{}^{i=1\ldots n},\ \mathsf{m}_j{:}\,\varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}] \\ \mathit{\Gamma} \vdash x{:}A{::}T_{\mathsf{res}}(x) \quad \mathit{\Gamma} \vdash y{:}A_k{::}T_{\mathsf{res}}(y) \end{array}}{\mathit{\Gamma} \vdash x.\mathsf{f}_k \mathrel{\mathsf{:=}} y{:}A{::}T_{\mathsf{upd}}(x,\mathsf{f}_k,y)}\ 1{\le}k{\le}n$$

The logic also provides a (structural) notion of subspecification, which generalises the usual notion of subtyping. So $\overline{x} \vdash A <: B$ holds if all fields of B are also fields of A with the same specification, and hereditarily all methods of B are methods of A with a stronger transition specification.

For instance, in the example in Tab. 2, $\vdash A_{\texttt{Manager}} <: [\texttt{rate} : \mathit{Int}]$ would be used in order to prove

$$m{:}A_{\texttt{Manager}}, x{:}A_{\texttt{AccFactory}} \vdash x.\texttt{manager:=}m : A_{\texttt{AccFactory}} :: T_{\mathsf{upd}}(x, \texttt{manager}, m)$$

when creating the reference to the manager object in the `manager` field of the factory object.

Semantics of Specifications. We give a denotational semantics of specifications. Each transition relation $\overline{x} \vdash T$ with free variables contained in $\overline{x}$ denotes a predicate

$$[\![\overline{x} \vdash T]\!]\rho \in \mathcal{P}(\mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}})$$

depending on an environment ρ. This can be defined in a straightforward way [18]. Observe that the meaning of a transition relation $\vdash T$ without free variables does not depend on the environment, and we sometimes simply write $[\![T]\!]$ in this case.

Similarly, an object specification $\overline{x} \vdash A$ gives rise to a predicate that depends on values for the free variables (since the underlying logic in the transition relations is untyped, the specifications of the free variables $\overline{x}$ are not relevant here). The interpretation of specifications

$$[\![\overline{x} \vdash A]\!]\rho \in \mathcal{P}(\mathsf{Val} \times \mathsf{St})$$

is given in Table 3. Subspecifications are simply set containment: If $\overline{x} \vdash A <: B$ then $[\![\overline{x} \vdash A]\!]\rho \subseteq [\![\overline{x} \vdash B]\!]\rho$.

4 Store Specifications

Object specifications are not sufficient. This is a phenomenon of languages with higher-order store well known from subject reduction and type soundness proofs (see [1–Ch. 11], [10]). Since statements may call subprograms residing in the store it has to be verified as well.

The standard remedy – also used in [2] – is to relativise the typing judgement such that it only needs to hold for “verified” stores. In other words, judgements

Table 3. Semantics of specifications

$$[\![\overline{x} \vdash \mathit{Bool}]\!]\rho = \mathsf{BVal} \times \mathsf{St}$$

$$[\![\overline{x} \vdash [\mathsf{f}_i \colon {A_i}^{i=1\ldots n},\ \mathsf{m}_j \colon \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]]\!]\rho =$$

$$\left\{ (l,\sigma) \in \mathsf{Loc} \times \mathsf{St} \;\middle|\; \begin{array}{ll} \textbf{(F)} & \text{for all } 1 \le i \le n.\ \sigma.l.\mathsf{f}_i \in [\![\overline{x} \vdash A_i]\!]\rho \\ \textbf{(M)} & \text{for all } 1 \le j \le m,\ \text{if } \sigma.l.\mathsf{m}_j(\sigma) = (v,\sigma')\!\downarrow \\ & \quad \text{then } (v,\sigma') \in [\![\overline{x}, y_j \vdash B_j]\!]\rho[y_j := l] \\ & \quad \text{and } (\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x}, y_j \vdash T_j]\!]\rho[y_j := l] \end{array} \right\}$$

are interpreted wrt. *store specifications.* A store specification assigns a specification to each location in a store. When an object is created, the specification assigned to it at the time of creation is included in the store specification.

In this section we will interpret such store specifications using the techniques from [19]. Since their denotations will rely on mixed-variant recursion, it is impossible to define a semantic notion of subspecification. Alas, the Abadi-Leino logic makes essential use of subspecifications. We get around this problem by only using a subset relationship on (denotations of) object specifications (where there is no contravariant occurrence of store as the semantics of objects is w.r.t. one fixed store, cf. Table 3).

Unfortunately, we are restricted by the logic's requirement that verified statements never break the validity of store specifications. In the case of field update this implies that subspecifications need to be invariant in their fields. As the semantic interpretation of the subspecification relation cannot reflect this, we were forced to sometimes use the syntactic subspecification relation.

Store Specifications and their Semantics. A store specification Σ assigns *closed* specifications to (a finite set of) locations:

Definition 1 (Store Specification). *A* store specification Σ *is a record* $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(\mathit{Spec})$ *s.t.* $\Sigma.l = A$ *implies* $\vdash A$. *For store specifications* Σ, Σ' *we say* Σ' extends Σ, *written* $\Sigma' \succcurlyeq \Sigma$, *if* $\Sigma.l = \Sigma'.l$ *for all* $l \in \mathit{dom}(\Sigma)$.

Because we focus on closed specifications in the following, we need a way to turn the components B_j of a specification $[\mathsf{f}_i \colon {A_i}^{i=1\ldots n},\ \mathsf{m}_j \colon \varsigma(y_j)B_j{::}{T_j}^{j=1\ldots m}]$ (recall that they may depend on y_j) into closed specifications. This is done by extending the syntax of expressions with locations: There is one symbol $\underline{l}$ for each $l \in \mathsf{Loc}$. When clear from context we will simply write l in place of $\underline{l}$. Further we write $A[\rho/\Gamma]$ for the simultaneous substitution of all $x \in [\Gamma]$ in A by $\underline{\rho(x)}$.

We can then abstract away from particular stores $\sigma \in \mathsf{St}$, and interpret closed result specifications $\vdash A$ with respect to such store specifications:

Definition 2 (Object Specifications). *For closed* A *let* $||A||_\Sigma \subseteq \mathsf{Val}$ *be*

$$\begin{aligned} ||\mathit{Bool}||_\Sigma &= \mathsf{BVal} \\ ||A||_\Sigma &= \{l \in \mathsf{Loc} \mid\ \vdash \Sigma.l <: A\} \end{aligned}$$

where $A \equiv [\mathsf{f}_i{:}\,A_i{}^{i=1\ldots n},\ \mathsf{m}_j{:}\,\varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}]$. *This extends to contexts in the natural way.*

Observe that for all A, if $\Sigma' \succcurlyeq \Sigma$ then $||A||_{\Sigma} \subseteq ||A||_{\Sigma'}$. We obtain the following lemma about *context extensions*.

Lemma 1 (Context Extension). *If* $\rho \in ||\Gamma||_{\Sigma}$, $\Gamma, x{:}A$ *is a well-formed context and* $v \in ||A[\rho/\Gamma]||_{\Sigma}$ *then* $\rho[x := v] \in ||\Gamma, x{:}A||_{\Sigma}$.

In light of the object introduction rule, we would like to interpret store specifications as predicates over stores, as follows.

$\sigma \in [\![\Sigma]\!] :\Leftrightarrow$
$\quad\forall l \in \mathsf{dom}(\Sigma)$ where $\Sigma.l = [\mathsf{f}_i{:}\,A_i{}^{i=1\ldots n},\ \mathsf{m}_j{:}\,\varsigma(y_j)B_j{::}T_j{}^{j=1\ldots m}]$:
$\quad\quad$**(F)** $\sigma.l.\mathsf{f}_i \in ||A_i||_{\Sigma}$ for all $1 \leq i \leq n$, and
$\quad\quad$**(M)** $\forall \Sigma' \succcurlyeq \Sigma\ \forall \sigma' \in [\![\Sigma']\!]\ \forall v \in \mathsf{Val}\ \forall \sigma'' \in \mathsf{St}$, if $\sigma.l.\mathsf{m}_j(\sigma') = (v, \sigma'')\!\downarrow$ then
$\quad\quad\quad$**(M1)** $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l/y_j]]\!]$
$\quad\quad\quad$**(M2)** $\exists \Sigma'' \succcurlyeq \Sigma'$ s.t. $\sigma'' \in [\![\Sigma'']\!]$
$\quad\quad\quad$**(M3)** $v \in ||B_j[l/y_j]||_{\Sigma''}$, for all $1 \leq j \leq m$

The universal quantification over extensions Σ' in **(M)** acccounts for (the specifications) of objects allocated between definition time and call time of methods. The existential quantification over extensions Σ'' in **(M2)** and **(M3)** provides for objects allocated by the method. In particular, since the result of a method call may be a freshly allocated object it is not sufficient to simply use Σ' in **(M2)** and **(M3)**. This semantic structure also appears in possible world models of other languages with dynamic allocation [10, 15].

Note the contravariant occurrence of $[\![-]\!]$ in $\forall \sigma' \in [\![\Sigma']\!]$ in **(M)**. Unfortunately, the usual techniques for establishing the existence of such predicates involving a mixed-variance recursion [12, 19] do not apply. They require the functional corresponding to the above recursion to map admissible predicates to admissible predicates. Due to the existential quantification in **(M2)** and **(M3)** this is not the case here.

We get around this problem by observing that the dynamic behaviour of programs (wrt. allocation of storage) can in fact be described more exactly, and the existential quantifier can be replaced: The elements of the (recursively defined) domain

$$\phi \in \mathsf{SF} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{St} \times \mathsf{SF} \times \mathit{Spec} \rightharpoonup \mathit{Spec} \times \mathsf{SF})) \tag{6}$$

are called *choice functions*, or *Skolem Functions*. The intuition is that, given a store $\sigma \in [\![\Sigma]\!]$, if $\sigma' \in [\![\Sigma']\!]$ with choice function ϕ', for some extension $\Sigma' \succcurlyeq \Sigma$ and the method invocation $\sigma.l.\mathsf{m}(\sigma')$ terminates, then $\phi.l.\mathsf{m}(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$ yields a store specification $\Sigma'' \succcurlyeq \Sigma'$ such that $\sigma'' \in [\![\Sigma'']\!]$ (and ϕ'' is a choice function for the extension Σ'' of Σ).

Using SF in the definition below has the effect of constraining the existential quantifier to work *uniformly* on the elements of increasing chains.

Definition 3 (Store Predicate). *Let $\mathcal{P} = \mathcal{P}(\mathsf{St} \times \mathsf{SF})^{\mathsf{Rec}_{\mathsf{Loc}}(Spec)}$ denote the collection of families of subsets of $\mathsf{St} \times \mathsf{SF}$, indexed by store specifications. We define a functional $\Phi : \mathcal{P}^{op} \times \mathcal{P} \to \mathcal{P}$ as follows.*

$(\sigma, \phi) \in \Phi(Y, X)_\Sigma :\Leftrightarrow$
(1) $\mathit{dom}(\Sigma) = \mathit{dom}(\phi)$ *and* $\forall l \in \mathit{dom}(\Sigma).\, \mathit{dom}(\pi_2(\Sigma.l)) = \mathit{dom}(\phi.l)$, *and*
(2) $\forall l \in \mathit{dom}(\Sigma)$ *where* $\Sigma.l = [f_i \colon A_i{}^{i=1\ldots n},\ m_j \colon \varsigma(y_j)B_j {::} T_j{}^{j=1\ldots m}]$:
(F) $\sigma.l.f_i \in ||A_i||_\Sigma$ *for all* $1 \le i \le n$, *and*
(M) $\forall \Sigma' \succcurlyeq \Sigma\ \forall(\sigma', \phi') \in Y_{\Sigma'}$. *if* $\sigma.l.m_j(\sigma') = (v, \sigma'')\downarrow$ *then*
(M1) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l/y_j]]\!]$
(M2) $\phi.l.m_j(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$ *s.t.* $\Sigma'' \succcurlyeq \Sigma'$ *and* $(\sigma'', \phi'') \in X_{\Sigma''}$
(M3) $v \in ||B_j[l/y_j]||_{\Sigma''}$ *for all* $1 \le j \le m$

We write $\sigma \in [\![\Sigma]\!]$ *if there is some* $\phi \in \mathsf{SF}$ *s.t.* $(\sigma, \phi) \in \mathit{fix}(\Phi)_\Sigma$.

Lemma 2. *Functional Φ, defined in Def. 3, does have a unique fixpoint.*

Proof. Firstly, one shows that Φ is monotonic and maps admissible predicates to admissible predicates, in the sense that for all X and Y,

$$\forall \Sigma.\ X_\Sigma \subseteq \mathsf{St} \times \mathsf{SF} \text{ admissible } \Rightarrow \forall \Sigma.\ \Phi(Y, X)_\Sigma \subseteq \mathsf{St} \times \mathsf{SF} \text{ admissible}$$

Next, define for all admissible $X, Y \in \mathcal{P}$, $e_1 \in [\mathsf{St} \rightharpoonup \mathsf{St}]$ and $e_2 \in [\mathsf{SF} \rightharpoonup \mathsf{SF}]$:

$$\langle e_1, e_2 \rangle : X \subset Y \textit{ iff } \forall \Sigma, \sigma, \phi.\ (\sigma, \phi) \in X_\Sigma \wedge \langle e_1, e_2 \rangle(\sigma, \phi)\downarrow \Rightarrow \langle e_1, e_2 \rangle(\sigma, \phi) \in Y_\Sigma$$

such that $e : X \subset Y$ states that $e = \langle e_1, e_2 \rangle$ maps pairs of stores and choice functions that are in X_Σ to pairs of stores and choice functions that are in corresponding component Y_Σ of Y. Let F be the locally continuous, mixed-variant functor associated with the domain equations (1) and (6), for which $F((\mathsf{St}, \mathsf{SF}), (\mathsf{St}, \mathsf{SF})) = (\mathsf{St}, \mathsf{SF})$ is the minimal invariant [12]. According to [12] it only remains to be shown that

$$e : X \subset X' \ \wedge\ e : Y' \subset Y \ \Rightarrow\ F(e, e) : \Phi(Y, X) \subset \Phi(Y', X') \qquad (\dagger)$$

for all $X, Y, X', Y' \in \mathcal{P}$ and $e \sqsubseteq id_{\mathsf{St} \times \mathsf{SF}}$ which follows from a similar line of reasoning as in [19]

Predicates denoting transition specifications must be upward-closed in the pre-execution store and downward-closed in the post-execution store. This holds in Abadi-Leino logic as transition specifications are only defined on the flat part of the store; if they referred to the method part, (†) could not necessarily be shown.

The next lemma establishes the relation between store and object specifications.

Lemma 3. *For all object specifications A, store specifications Σ, stores σ, and locations l, if $\sigma \in [\![\Sigma]\!]$ and $l \in \mathit{dom}(\Sigma)$ such that $\vdash \Sigma.l <: A$ then $(l, \sigma) \in [\![A]\!]$.*

5 Soundness

We can now define the semantics of judgements of Abadi-Leino logic and prove the key lemma.

Definition 4 (Validity). *$\Gamma \models a : A :: T$ if and only if for all store specifications $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Spec)$, for all $\rho \in ||\Gamma||_{\Sigma}$ and all $\sigma \in [\![\Sigma]\!]$, if $[\![a]\!]\rho\sigma = (v, \sigma')$ then $(v, \sigma') \in [\![[\Gamma] \vdash A]\!]\rho$ and $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T]\!]\rho$.*

Lemma 4 (Soundness and Invariance). *Suppose*
(H1) $\Gamma \vdash a : A :: T$
(H2) $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Spec)$ *is a store specification*
(H3) $\rho \in ||\Gamma||_{\Sigma}$
Then there exists $\phi \in [\mathsf{St} \times \mathsf{SF} \times Spec \rightharpoonup Spec \times \mathsf{SF}]$ s.t. for all $\Sigma' \succcurlyeq \Sigma$ and for all $(\sigma', \phi') \in \mathit{fix}(\Phi)_{\Sigma'}$, if $[\![a]\!]\rho\sigma' = (v, \sigma'')\downarrow$ then the following holds:
(S1) *there exists $\Sigma'' \succcurlyeq \Sigma'$ and $\phi'' \in \mathsf{SF}$ s.t. $\phi(\sigma', \phi', \Sigma') = (\Sigma'', \phi'')$*
(S2) $(\sigma'', \phi'') \in \mathit{fix}(\Phi)_{\Sigma''}$
(S3) $v \in ||A[\rho/\Gamma]||_{\Sigma''}$
(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T]\!]\rho$

Note that condition **(S1)** explicates that store specifications are preserved by the execution of proved programs, which allows the inductive proof to go through.

Proof. The proof is by induction on the derivation of $\Gamma \vdash a : A :: T$ (whereas the original proof [2] is by induction on the length of the computation). Generally, we have to distinguish the case of objects, which are stored in the heap, and Booleans, which are stack allocated.

- Lemma 1 is applied in the cases (let) and (object construction), where an extended specification context is used in the induction hypothesis.
- Invariance of the field components in subspecifications is needed in the case for (field update).
- In the cases where the store changes, i.e., (object construction) and (field update), we must show explicitly that the resulting store satisfies the store specification, according to Definition 3. This is tedious but not difficult, due to the definition of $\sigma \in [\![\Sigma]\!]$.

Lemma 3 and Lemma 4 immediately prove

Theorem 1 (Soundness). *If $\Gamma \vdash a : A :: T$ then $\Gamma \models a : A :: T$.*

6 Denotational Analysis of Abadi-Leino Logic

For the proof of Theorem 2, establishing the existence of store predicates, it is necessary that transition relations are upwards and downwards closed in their first and second store argument, respectively. A sufficient condition is that transition relations work on the flat part of stores only. This provides an explanation why the transition relations of the Abadi-Leino do not refer to methods.

6.1 Extensions

This section contains a list of possible extensions of the Abadi-Leino logic. We think that the denotational semantics helps to clarify their feasibility.

Invariants of Fields. Abadi and Leino's logic is peculiar in that verified programs need to preserve store specifications. Put differently, only properties which are in fact preserved can be expressed in object specifications. In particular, specifying fields in object specifications is limited. Invariants like e.g. $\texttt{balance} \geq 0$, stating that an account comes without overdraft, cannot be formulated. The same axiom in a transition specification would only guarantee that the actual balance is positive. For "private" (local) fields, invisible to other objects, such invariants can be easily accommodated.

Method Parameters. Formal method parameters of the form $x : A$ can be attached to method specifications, e.g.,

$$\texttt{deposit}(x : \mathit{Int}) : \varsigma(y)[] :: \exists z.z = \grave{\sigma}(y, \texttt{balance}) \wedge T_{\mathsf{upd}}(y, \texttt{balance}, z + x)$$

by adding an extra assumption to the definition of store specifications. When $\sigma' \in [\![\Sigma']\!]$ then **(M1)**–**(M3)** have to be shown for all $v \in ||A||_{\Sigma'}$ where v is the actual parameter replacing formal parameter x in the method call.

Dynamic Loading. Dynamic loading of objects is, in a way, already available in the object calculus (this is one of its advantages over class-based languages). Loading an object of which only its specification $A \equiv [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(x_j)B_j{::}T_j]$ is known corresponds to using a command of which one only knows its result specification A. Thus, $x : [m : \varsigma(y)A :: \exists \overline{z}.\ T_{\mathsf{obj}}(\mathsf{f}_i = z_i)] \vdash x.m() : A :: \exists \overline{z}.\ T_{\mathsf{obj}}(\mathsf{f}_i = z_i)$ describes dynamic loading where the load command is $x.m()$. It can be used to load any object fulfilling specification A.

Parametric Method Specifications. Transition specifications cannot refer to methods. While this is adequate when all method specifications are known it prevents verification of programs that use delegations (similar to the Command pattern). The flatness of transition relations is sufficient but not necessary for the existence of store specifications. Therefore "parametric" method specifications may be possible.

Method Update. Although method update is not allowed in Abadi-Leino logic, fields can be updated and thus the methods in a field object (similar to the Decorator pattern). By the invariance of object specifications, the object used for the update must satisfy the specification of the field to be updated. Any extra conditions that the new object may fulfil are not recorded and cannot be used later. More useful would be a "behavioural" update where result and transition specifications of the overriding method are subspecifications of the original method. This seems to be impossible as there is no notion of subspecification for store specifications.

Recursive Specifications. Recursive specifications are necessary when a field of an object or a result of one of the object's methods are supposed to satisfy the same specification as the object itself. They are needed to specify any recursive datatype. For example, if $A_{\texttt{Manager}}$ should include a list of accounts, we would need a recursive specification $\mu X.\ [\mathsf{head} : A_{\texttt{Account}}, \mathsf{tail} : X]$.

Below we discuss in more detail how recursive specifications can be dealt with in the logic.

6.2 Recursive Specifications

Syntax and Proof Rules. We introduce recursive specifications $\mu(X)A$. To prevent meaningless specifications such as $\mu(X)X$ we only allow recursion through object specifications, thereby enforcing "formal contractiveness".

$$\begin{aligned} \underline{A} &::= \top \mid \mathit{Bool} \mid [\mathsf{f}_i{:}\, A_i{}^{i=1\dots n},\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j{}^{j=1\dots m}] \mid \mu(X)\underline{A} \\ A, B &::= \underline{A} \mid X \end{aligned}$$

where X ranges over an infinite set *TyVar* of specification variables. X is bound in $\mu(X)A$, and as usual we identify specifications up to the names of bound variables.

In addition to specification contexts Γ we introduce contexts Δ that contain specification variables with an upper bound, $X <: A$, where A is either another variable or $\top$. In the rules of the logic we replace $\Gamma \vdash \dots$ by $\Gamma; \Delta \vdash \dots$, and the definitions of well-formed specifications and well-formed specification contexts are extended, similar to the case of recursive types [1].

Subspecifications for recursive specifications are obtained by the "usual" recursive subtyping rule [3],

$$\frac{\Gamma; \Delta, Y <: \top, X <: Y \vdash A <: B}{\Gamma; \Delta \vdash \mu X.A <: \mu Y.B}$$

As will be seen from the semantics below, in our model a recursive specification and its unfolding are not just isomorphic but equal, i.e., $[\![\mu X.A]\!] = [\![A[(\mu X.A)/X]]\!]$. Hence we can add $\Gamma; \Delta \vdash A[(\mu X.A)/X] <: \mu X.A$ and $\Gamma; \Delta \vdash \mu X.A <: A[(\mu X.A)/X]$ and do not need to introduce *fold* and *unfold* terms.

Semantics of Recursive Specifications. We extend the interpretation of specifications to the new cases, where η maps type variables to admissible subsets of $\mathsf{Val} \times \mathsf{St}$:

$$\begin{aligned} [\![\Gamma; \Delta \vdash \top]\!]\rho\eta &= \mathsf{Val} \times \mathsf{St} \\ [\![\Gamma; \Delta \vdash X]\!]\rho\eta &= \eta(X) \\ [\![\Gamma; \Delta \vdash \mu(X)A]\!]\rho\eta &= \mathsf{gfp}(\lambda\chi.[\![\Gamma; \Delta, X <: \top \vdash A]\!]\rho\eta[X = \chi]) \end{aligned}$$

We write $\eta \models \Delta$ if, for all $X <: Y$ in Δ, $\eta(X) \subseteq \eta(Y)$. The set of admissible subsets of $\mathsf{Val} \times \mathsf{St}$ is closed under arbitrary intersections, hence forms a complete lattice when ordered by set inclusion, as do environments η with the point-wise ordering $\leq$. Using well-known facts about lattices and monotonic maps one observes that the semantics preserves meets:

$$\eta_0 \geq \eta_1 \geq \ldots \Rightarrow [\![\Gamma; \Delta \vdash A]\!]\rho(\bigwedge_i \eta_i) = \bigcap_i [\![\Gamma; \Delta \vdash A]\!]\rho\eta_i$$

In particular, the greatest fixed point in the interpretation above is guaranteed to exist, as $\mathsf{gfp}(f) = \bigcap_i f^i(\top)$ for monotonic and meet preserving f.

Existence of Store Predicates. Next, we adapt our notion of store specification to recursive specifications. A store specification is now taken to be a record $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(\mathit{Spec})$ such that $\Sigma.l = \mu(X)[\mathsf{f}_i : A_i{}^{i=1\ldots n}, \mathsf{m}_j : \varsigma(y_j)B_j :: T_j{}^{j=1\ldots m}]$ is a closed (recursive) object specification, for each $l \in \mathsf{dom}(\Sigma)$. Because of the (fold) and (unfold) rules, the requirement that only object specifications with a μ-binder in head position occur in Σ is no real restriction. The definition of the functional Φ of Section 4 remains virtually the same apart from an unfolding of the recursive specification in the cases for field and method result specification:

$(\sigma, \phi) \in \Phi(R, S)_\Sigma :\Leftrightarrow$
 (1) ...
 (2) $\forall l \in \mathsf{dom}(\Sigma)$ where $\Sigma.l = \mu(X)[\mathsf{f}_i : A_i{}^{i=1\ldots n}, \mathsf{m}_j : \varsigma(y_j)B_j :: T_j{}^{j=1\ldots m}]$:
  (F) $\sigma.l.\mathsf{f}_i \in ||A_i[\Sigma.l/X]||_\Sigma$
   ...
  (M3) $v \in ||B_j[\Sigma.l/X, l/y_j]||_{\Sigma''}$
   ...

The proof of Lemma 2 can be easily adapted to show that this functional also has a unique fixpoint.

Syntactic Approximations. In Section 5, Lemma 3 was proved by induction on the structure of A. This inductive proof cannot be extended directly to prove a corresponding result for recursive specifications: The recursive unfolding in cases **(F)** and **(M3)** of the definition of $\sigma \in [\![\Sigma]\!]$ would force a similar unfolding of A in the inductive step. We consider finite approximations as in [3], where we get rid of recursion by unfolding a finite number of times and replacing all remaining occurrences of recursion by $\top$.

Definition 5 (Approximations). *For each A and $k \in \mathbb{N}$, we define $A|^k$ as*

- $A|^0 = \top$
- $\mu(X)A|^{k+1} = A[\mu(X)A/X]|^{k+1}$
- $[f_i : A_i{}^{i=1\ldots n}, m_j : \varsigma(y_j)B_j :: T_j{}^{j=1\ldots m}]|^{k+1} =$ $[(f_i : A_i|^k)^{i=1\ldots n}, m_j : \varsigma(y_j)B_j|^k :: T_j{}^{j=1\ldots m}]$
- $\top|^{k+1} = \top$
- $X|^{k+1} = X$
- $\mathit{Bool}|^{k+1} = \mathit{Bool}$

Lemma 5. *For all $\Gamma; \Delta \vdash A$ and $\eta \models \Delta$, $[\![\Gamma; \Delta \vdash A]\!]\rho\eta = \bigcap_{k \in \mathbb{N}} [\![\Gamma; \Delta \vdash A|^k]\!]\rho\eta$.*

Lemma 6. *For all $\sigma \in [\![\Sigma]\!]$, $l \in \mathsf{dom}(\Sigma)$ and (possibly recursive) A s.t. $\vdash \Sigma.l <: A$, $(l, \sigma) \in [\![A]\!]$.*

Proof. Similar to the proof of Lemma 3 one shows $(l, \sigma) \in [\![A|^k]\!]$ for all k. Then by Lemma 5, $(l, \sigma) \in \bigcap_{k \in \mathbb{N}} [\![A|^k]\!] = [\![A]\!]$.

7 Conclusion

Based on a denotational semantics, we have given a soundness proof for Abadi and Leino's program logic of an object-based language. Compared to the original proof, which was carried out wrt. an operational semantics, our techniques allowed us to distinguish the notions of derivability and validity. Further, we used the denotational framework to extend the logic to recursive object specifications. In comparison to a similar logic presented in [9] our notion of subspecification is structural rather than nominal.

Although our proof is very much different from the original one, the nature of the logic forces us to work with store specifications too. Information for locations referenced from the environment Γ will be needed for derivations. Since the Γ cannot reflect the dynamic aspect of the store (which is growing) one uses store specifications Σ. They do not show up in the Abadi-Leino logic as they are automatically preserved by programs. By contrast to [2], we can view store specifications as predicates on stores which need to be defined by mixed-variant recursion due to the form of the object introduction rule. Unfortunately, such recursively defined predicates do not directly admit an interpretation of subsumption (nor weakening).

Conditions **(M1)** – **(M3)** in the semantics of store specifications ensure that methods in the store preserve not only the current store specification but also arbitrary extensions $\Sigma' \succcurlyeq \Sigma$. Clearly, not every predicate on stores is preserved. As we lack a semantic characterisation of those specifications that are syntactically definable (as Σ), specification syntax appears in the definition of $\sigma \in [\![\Sigma]\!]$ (Def. 3). More annoyingly, field update requires subspecifications to be invariant in the field components. We do not know how to express this property of object specifications semantically (on the level of predicates) and need to use the inductively defined subspecification relation instead.

The proof of Lemma 2, establishing the existence of store predicates, provides an explanation why transition relations of the Abadi-Leino logic express properties of the flat part of stores only and allows for a quick check whether extensions are feasible. We have enumerated several extensions in Section 6.1. Based on this list and the results presented we intend to design a variation of the Abadi-Leino logic that is more expressive. We hope that this will also shed some light on modular reasoning for class-based languages.

Acknowledgement. We wish to thank Thomas Streicher for discussions and comments.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, New York, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In N. Dershowitz, editor, *Verification: Theory and Practice*, pages 11–41. Springer, 2004.
3. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

4. K. R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
5. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *FOSSACS'99*, volume 1578 of *LNCS*, pages 135–149, 1999.
6. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 105–121, 1998.
7. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
8. M. Hofmann and F. Tang. Generation of verification conditions for Abadi and Leino's logic of objects. Presented at 9th International Workshop on Foundations of Object-Oriented Languages, 2002.
9. K. R. M. Leino. Recursive object types in a logic of object-oriented programs. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 170–184, 1998.
10. P. B. Levy. Possible world semantics for general storage in call-by-value. In J. Bradfield, editor, *CSL'02*, volume 2471 of *LNCS*. Springer, 2002.
11. L. C. Paulson. *Logic and Computation : Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. 1987.
12. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
13. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 162–176, 1999.
14. U. S. Reddy. Objects and classes in algol-like languages. *Information and Computation*, 172(1):63–97, 2002.
15. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, 2004.
16. B. Reus. Class-based versus object-based: A denotational comparison. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST'02*, volume 2422 of *LNCS*, pages 473–488, 2002.
17. B. Reus. Modular semantics and logics of classes. In M. Baatz and J. A. Makowsky, editors, *CSL'03*, volume 2803 of *LNCS*, pages 456–469. Springer Verlag, 2003.
18. B. Reus and J. Schwinghammer. Denotational semantics for Abadi and Leino's logic of objects. Technical Report 2004:03, Informatics, University of Sussex, 2004.
19. B. Reus and T. Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316:191–213, 2004.
20. B. Reus, M. Wirsing, and R. Hennicker. A Hoare-Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In H. Hussmann, editor, *FASE 2001*, volume 2029 of *LNCS*, pages 300–317, Berlin, 2001. Springer.
21. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.