A Step-indexed Semantics of Imperative Objects

Cătălin Hriţcu
Department of Computer Science
Saarland University
hritcu@cs.uni-sb.de

Jan Schwinghammer
Programming Systems Lab
Saarland University
jan@ps.uni-sb.de

Abstract

Step-indexed semantic models of types were proposed as an alternative to purely syntactic proofs of type safety using subject reduction. Building on work by Ahmed, Appel and others, we introduce a step-indexed model for the imperative object calculus of Abadi and Cardelli. Providing a semantic account of this calculus using more 'traditional', domain-theoretic approaches has proved challenging due to the combination of dynamically allocated objects, higher-order store, and an expressive type system. Here we show that the step-indexed model can interpret a rich type discipline with object types, subtyping, recursive and bounded quantified types in the presence of state.

Keywords Formal calculi, type systems, language semantics

1. Introduction

The *imperative object calculus* of Abadi and Cardelli is a very small, yet very expressive object-oriented language [2]. Despite the extreme simplicity of its syntax, the calculus models many important concepts of object-oriented programming, as well as the often subtle interaction between them. In particular it raises interesting and non-trivial questions with respect to typing.

In contrast to the more common class-based object-oriented languages, in the imperative object calculus every object comes equipped with its own set of methods which can be updated at runtime. As a consequence, the methods need to reside in the store, i.e. the store is *higher-order*. Moreover, objects are *allocated dynamically* and aliasing is possible. Dynamically-allocated, higher-order store is present in different forms in many practical programming languages (*e.g.* pointers to functions in C and general references in SML), but it considerably complicates the construction of adequate semantic models in which one can reason about the behaviour of programs (*cf.* [29]).

Purely syntactic arguments such as subject-reduction suffice for proving the soundness of traditional type systems. However, once such type systems are turned into powerful specification languages, like the logic of objects of Abadi and Leino [4] or the hybrid type system of Flanagan *et al.* [18], arguments based solely on the operational semantics seem no longer appropriate. The meaning of assertions is no longer obvious, since they have to describe the code on the heap. We believe that specifications of program behaviour should have a meaning independent of the particular proof system on which syntactic preservation proofs rely, as also argued by [13, 30].

In the setting described above one would ideally prove soundness with respect to a semantic model that makes a clear distinction between semantic validity and derivability using the syntactic rules. However, building such semantic models is challenging, and there is currently no fully satisfactory semantic account of the imperative object calculus:

Denotational semantics. Domain-theoretic models have been employed in proving the soundness of the logic of Abadi and Leino [30, 31]. However, the existing techniques fall short of providing convincing models of *typed* objects: [31] considers an untyped semantics, and the model of [30] handles neither second-order types, nor object types with variance annotations. Due to the dynamic-allocated higher-order store present in the imperative object calculus, the models rely on techniques for recursively defined domains in functor categories [24, 28]. This makes them complex, and establishing properties even for specific programs often requires a substantial effort.

Equational reasoning. Gordon *et al.* develop reasoning principles for establishing the contextual equivalence of untyped objects, and apply them to prove correctness of a compiler optimization [19]. Jeffrey and Rathke consider a concurrent variant of the calculus, and characterize may-testing equivalence in terms of the trace sets generated by a labeled transition system [22]. In both cases the semantics is limited to equational reasoning, *i.e.* establishing contextual equivalences between programs. In theory, this can be used to verify a program by showing it equivalent to one that is trivially correct and acts as a specification. However, this can be more cumbersome in practice than using program logics, the established formalism for specifying and proving the correctness of programs.

Translations. Abadi *et al.* [3] give an adequate encoding of the imperative object calculus into a lambda calculus with records, references, recursive and existential types and subtyping. Together with an interpretation of this target language, an adequate model for the imperative object calculus could, in principle, be obtained. However, we are not aware of any adequate domain-theoretic model for general references and impredicative second-order types. And, even if such a model would be constructed, it will still be preferable to have a self-contained model for the imperative object calculus, without the added complexity of the (non-trivial) translation.

A solution to this problem could be the step-indexed semantic models introduced by Appel *et al.* as an alternative to subject-reduction proofs [9, 10]. Such models are based directly on the operational semantics, and are much simpler than the existing domain-theoretic models. In this setting the types are simply interpreted as sets of syntactic values indexed by a number of computation steps. Intuitively, a term belongs to a certain type if it behaves like an element of that type for any number of steps. Every type is built as a sequence of increasingly accurate semantic approximations, which allows one to easily deal with recursion. Type safety is an immediate consequence of this interpretation of types, and the semantic counterparts of the usual typing rules are proved as independent lemmas, either directly or by induction on the index. Ahmed *et al.* successfully applied this generic technique

$$
\begin{array}{lll}
A, B ::= X \mid \mathit{Top} \mid \mathit{Bot} \mid A \rightarrow B & & \text{(types)} \\
\quad \mid [\mathrm{m}_d :_{\nu_d} A_d]_{d \in D} \mid \mu(X)A & & \\
\quad \mid \forall(X \leqslant A)B \mid \exists(X \leqslant A)B & & \\
\nu ::= \circ \mid + \mid - & & \text{(variance annot.)} \\
a, b ::= x & & \text{(variables)} \\
\quad \mid [\mathrm{m}_d = \varsigma(x_d{:}A)b_d]_{d \in D} & & \text{(object creation)} \\
\quad \mid a.\mathrm{m} & & \text{(method invocation)} \\
\quad \mid a.\mathrm{m} := \varsigma(x{:}A)b & & \text{(method update)} \\
\quad \mid \text{clone } a & & \text{(shallow copy)} \\
\quad \mid \lambda(x{:}A)b & & \text{(procedures)} \\
\quad \mid a\ b & & \text{(application)} \\
\quad \mid \mathrm{fold}_A\ b & & \text{(recursive folding)} \\
\quad \mid \mathrm{unfold}_A\ b & & \text{(recursive unfolding)} \\
\quad \mid \Lambda(X \leqslant A)b & & \text{(type abstraction)} \\
\quad \mid a[A] & & \text{(type application)} \\
\quad \mid \text{pack } X \leqslant A = C \text{ in } a : B & & \text{(existential package)} \\
\quad \mid \text{open } a \text{ as } X \leqslant A, x{:}B \text{ in } b : C & & \text{(package opening)}
\end{array}
$$

Figure 1. Syntax of types and terms

to a lambda calculus with general references, impredicative polymorphism and recursive types [5, 7, 8].

In this paper we further extend the model of Ahmed *et al.* with object types and subtyping, and we use the resulting model to prove the soundness of an expressive type system for the imperative object calculus. The main contribution of our work is the novel semantics of object types.

Even though in this paper we are concerned with the safety of a type system, the step-indexing technique is not restricted to types, and has already been used for equational reasoning [6, 10] and for proving the soundness of Hoare-style program logics of low-level languages [13, 14]. We hope that eventually it will become possible to use a step-indexed model to prove the soundness of more expressive program logics for the imperative object calculus.

Outline The next section introduces the syntax, the operational semantics and the type system that we consider for the imperative object calculus. In Section 3 we present the step-indexed semantic model for this calculus. In particular, we provide the definitions of our semantic types together with the properties that they fulfill. In Section 4 these properties are used to prove the soundness of the initial type system. Section 5 concludes with a comparison to related work, together with some interesting directions for further investigation.

Full proofs can be found in the extended version of this paper [21].

2. The Imperative Object Calculus

We recall the syntax of the imperative object calculus with recursive and second-order types, and introduce a small-step operational semantics for this calculus that is equivalent to the big-step semantics given by Abadi and Cardelli [2].

2.1 Syntax

Let *Var*, *TVar* and *Meth* be pairwise disjoint, countably infinite sets of *variables*, *type variables* and *method names*, respectively. Let

$$
\begin{array}{l}
\mathcal{E}[\cdot] ::= [\cdot] \mid \mathcal{E}.\mathrm{m} \mid \mathcal{E}.\mathrm{m} := \varsigma(x{:}A)b \mid \text{clone } \mathcal{E} \mid \\
\quad \mid \mathcal{E}\ b \mid v\ \mathcal{E} \mid \mathrm{fold}_A\, \mathcal{E} \mid \mathrm{unfold}_A\, \mathcal{E} \mid \mathcal{E}[A] \\
\quad \mid \text{pack } X \leqslant A = C \text{ in } \mathcal{E} : B \mid \text{open } \mathcal{E} \text{ as } X \leqslant A, x{:}B \text{ in } b : C
\end{array}
$$

Figure 2. Evaluation contexts

x, y range over *Var*, X, Y range over *TVar*, and let m range over *Meth*. Figure 1 defines the syntax of the types and terms of the imperative object calculus.

Objects are unordered collections of named methods. In a method $\mathrm{m} = \varsigma(x{:}A)b$, ς is a binder that binds the 'self' argument x in the method body b. The self argument can be used inside the method body for invoking the methods of the containing object. Methods with arguments other than self can be obtained by having a procedural abstraction as the method body. The methods of an object can be invoked or updated, but no new methods can be added, and the existing methods cannot be deleted. The type of objects with methods named m_d that return results of type A_d, for d in some set D, is written as $[\mathrm{m}_d :_{\nu_d} A_d]_{d \in D}$, where $\nu \in \{\circ, +, -\}$ is a *variance annotation* that indicates if the method is considered *invoke-only* ($+$), *update-only* ($-$), or may be used without restriction ($\circ$).

We write procedural abstractions with type $A \rightarrow B$ as $\lambda(x{:}A)b$ and applications as $a\ b$, respectively. Although procedural abstractions can be defined as syntactic sugar in Abadi and Cardelli's calculus, it smoothes the theory in Section 3 to include them as primitives. We use fold_A and unfold_A to denote the isomorphism between a recursive type $\mu(X)B$ and its unfolding $\{\!\{X \mapsto \mu(X)B\}\!\}(B)$. Finally, we consider bounded universal and existential types $\forall(X \leqslant A)B$ and $\exists(X \leqslant A)B$ along with their introduction and elimination forms.

The set of free variables of a term a is denoted by $\mathit{fv}(a)$, and similarly the free type variables in a type A by $\mathit{fv}(A)$. We identify types and terms up to the consistent renaming of bound variables. We use $\{\!\{t \mapsto r\}\!\}$ to denote the singleton map that maps t to r. For a finite map σ from variables to terms, $\sigma(a)$ denotes the result of capture-avoiding substitution of all $x \in \mathit{fv}(a) \cap \mathit{dom}(\sigma)$ by $\sigma(x)$. The same notation is used for the substitution of type variables. Generally, for any function f, the notation $f[t := r]$ denotes the function that maps t to r, and otherwise agrees with f.

2.2 Operational Semantics

Let *Loc* be a countably infinite set of *heap locations* ranged over by l. We extend the set of terms by run-time representations of objects $\{\mathrm{m}_d{=}l_d\}_{d \in D}$, associating heap locations to a set of method names. *Values* are generated by the following grammar:

$$
\begin{array}{rl}
v \in \mathit{Val} ::= & \{\mathrm{m}_d{=}l_d\}_{d \in D} \mid \lambda(x{:}A)b \mid \mathrm{fold}_A\, v \\
\mid & \Lambda(X \leqslant A)b \mid \text{pack } X \leqslant A = C \text{ in } v : B
\end{array}
$$

Apart from run-time objects, values consist of procedures, type abstractions and existential packages as in the call-by-value lambda calculus. We often only consider terms and values without free variables, and denote the set of these *closed terms* and *closed values* by *CTerm* and *CVal*, respectively. We call *programs* the closed terms in which locations do not occur, and we denote the set of all programs by *Prog*. A *heap* h is a finite map from *Loc* to *CVal* [1], and we write *Heap* for the set of all heaps.

Figure 2 defines the set of *evaluation contexts*, formalizing a left-to-right, call-by-value strategy. We write $\mathcal{E}[a]$ for the term obtained by plugging a into the hole $[\cdot]$ of $\mathcal{E}$. The one-step reduc-

[1] In fact, for the purpose of modelling the object calculus it would suffice to regard procedures as the only kind of storable value.

(Red-Obj)	$\langle h, [\mathrm{m}_d{=}\varsigma(x_d{:}A)b_d]_{d\in D}\rangle \to \langle h\,[l_d := \lambda(x_d{:}A)b_d]_{d\in D}\,, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle$	where $\forall d \in D.\ l_d \notin \mathit{dom}(h)$
(Red-Inv)	$\langle h, \{\mathrm{m}_d{=}l_d\}_{d\in D}\,.\mathrm{m}_e\rangle \to \langle h, h(l_e)\ \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle$	if $e \in D$
(Red-Upd)	$\langle h, \{\mathrm{m}_d{=}l_d\}_{d\in D}\,.\mathrm{m}_e := \varsigma(x{:}A)b\rangle \to \langle h\,[l_e := \lambda(x{:}A)b]\,, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle$	if $e \in D$
(Red-Clone)	$\langle h, \text{clone}\ \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \to \langle h\,[l'_d := h(l_d)]_{d\in D}\,, \{\mathrm{m}_d{=}l'_d\}_{d\in D}\rangle$	where $\forall d \in D.\ l'_d \notin \mathit{dom}(h)$
(Red-Beta)	$\langle h, (\lambda(x{:}A)b)\ v\rangle \to \langle h, \{\!\!\{x \mapsto v\}\!\!\}(b)\rangle$	
(Red-Unfold)	$\langle h, \text{unfold}_A\ (\text{fold}_B\ v)\rangle \to \langle h, v\rangle$	
(Red-TBeta)	$\langle h, (\Lambda(X{\leqslant}A)b)[B]\rangle \to \langle h, \{\!\!\{X \mapsto B\}\!\!\}(b)\rangle$	
(Red-Open)	$\langle h, \text{open}\ v\ \text{as}\ X{\leqslant}A, x{:}B\ \text{in}\ b : C\rangle \to \langle h, \{\!\!\{x \mapsto v', X \mapsto C'\}\!\!\}(b)\rangle$	where $v \equiv \text{pack}\ X'{\leqslant}A' = C'\ \text{in}\ v' : B'$

Figure 3. One-step reduction

tion relation $\to$ is defined as the least relation on *configurations* $\langle h, a\rangle \in \mathit{Heap} \times \mathit{CTerm}$ generated by the rules in Figure 3 and closed under the following context rule:

$$\text{(Red-Ctx)} \qquad \langle h, a\rangle \to \langle h', a'\rangle \implies \langle h, \mathcal{E}[a]\rangle \to \langle h', \mathcal{E}[a']\rangle$$

The methods are actually stored in the heap as procedures. Object construction allocates new heap storage for these procedures and returns a record of references to them (Red-Obj). Upon method invocation the corresponding stored procedure is retrieved from the heap and applied to the enclosing object (Red-Inv). The self parameter is thus passed just like any other procedure argument. This makes the 'self-application' semantics of method invocation explicit, while technically, it allows us to more directly use the step-indexed model of Ahmed *et al.* [5, 7, 8].

While variables are immutable identifiers, methods can be updated destructively. Such updates only modify the heap and leave the run-time object unchanged (Red-Upd). Object cloning generates a shallow copy of the object in the heap (Red-Clone). The last four rules in Figure 3 are as in the lambda calculus.

For $k \in \mathbb{N}$, $\to^k$ denotes the k-step reduction relation. We write $\langle h, a\rangle \not\to$ if the configuration $\langle h, a\rangle$ is irreducible (*i.e.* there exists no configuration $\langle h', a'\rangle$ such that $\langle h, a\rangle \to \langle h', a'\rangle$).

Note that reduction is not deterministic, due to the arbitrarily chosen new locations in (Red-Obj) and (Red-Clone). However, we still have that there is always at most one, uniquely determined redex.

Proposition 2.1 (Unique decomposition)**.** *If $\langle h, a\rangle \to \langle h', a'\rangle$ then there exists a unique context $\mathcal{E}$, and two unique subterms b and b' such that $a \equiv \mathcal{E}[b]$, $\langle h, b\rangle \to \langle h', b'\rangle$ is an instance of one of the rules in Figure 3, and $a' \equiv \mathcal{E}[b']$.*

This has the important consequence that the reduction order is fixed, e.g., if there is a reduction sequence beginning with an application, $\langle h, a\ b\rangle \to^k \langle h', v\rangle$, then this sequence splits into

$$\langle h, a\ b\rangle \to^{i_1} \langle h_1, v_1\ b\rangle \to^{i_2} \langle h_2, v_1\ v_2\rangle \to^{k-i_1-i_2} \langle h', v\rangle$$

where $\langle h, a\rangle \to^{i_1} \langle h_1, v_1\rangle$ and $\langle h_1, b\rangle \to^{i_2} \langle h_2, v_2\rangle$ for some $i_1, i_2 \geq 0$. Similar decompositions into subsequences hold for reductions starting from the other term forms.

It is easy to see that the operational semantics is independent of the type annotations inside terms. Also the semantic types that we consider in Section 3 will not depend on the syntactic type expressions in the terms. In order to reduce the notational overhead and to prevent confusion between the syntax and semantics of types we will omit type annotations when presenting the step-indexed model. For example, instead of $a[A]$ we will merely write $a[]$.

2.3 Type System

The type system we consider features procedure, object, iso-recursive and impredicative, bounded quantified types, as well as subtyping, and corresponds to $\mathbf{FOb}_{<:\mu}$ from [2].

It is fairly standard and consists of four inductively defined typing judgments:

- $\Gamma \vdash \diamond$, describing *well-formed typing contexts*,
- $\Gamma \vdash A$, defining *well-formed types*,
- $\Gamma \vdash A \leqslant B$, for *subtyping* between well-formed types, and
- $\Gamma \vdash a : A$, for *typing terms*.

The typing context Γ is a list containing type bindings for the (term) variables $x{:}A$ and upper bounds for the type variables $X{\leqslant}A$. A typing context is well-formed if it does not contain duplicate bindings for (term or type) variables and all types appearing in it are well-formed. A type is well-formed with respect to a well-formed context Γ if all its type variables appear in Γ.

Figure 4 defines the subtype relation. It allows subtyping in width for the object types: an object type with more methods is a subtype of an object type with less methods, as long as the types of the common methods agree. For the invoke-only $(+)$ and update-only methods $(-)$ in a type, covariant respectively contravariant subtyping in depth is allowed (SubObj). Furthermore, the unrestricted methods $(\circ)$ can be regarded by subtyping as either invoke-only or update-only (SubObjVar). Since the annotations can be conveniently chosen at creation time (Obj) this brings much flexibility. As explained in [2], it allows the type system to distinguish between the invocations and updates done through the self argument, and the ones done from the outside.

Finally, Figure 5 defines the typing relation. The applicability of the rules for method invocation (Inv), and for method update (Upd), depends on the variance annotation. Also notice that only type-preserving updates are allowed (Upd). It is important to note that we do not give types to heap locations. The type system is simpler since it only checks programs, not partially evaluated terms as it would be required by a subject-reduction proof.

3. A Step-indexed Semantics of Objects

Modelling higher-order store is necessarily more involved than the treatment of first-order storage since the semantic domains become mutually recursive. Recall that heaps store values which may be procedures. These in turn can be modeled as functions that take a value and the initial heap as input, and return a value and the possibly modified heap upon termination. This suggests the

Subtyping $\boxed{\Gamma \vdash A \leqslant B}$

$$(\textsc{SubVar})\ \frac{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash \diamond}{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash X \leqslant A}$$

$$(\textsc{SubProc})\ \frac{\Gamma \vdash A' \leqslant A \quad \Gamma \vdash B \leqslant B'}{\Gamma \vdash A \to B \leqslant A' \to B'}$$

$$(\textsc{SubObj})\ \frac{\begin{array}{r}E \subseteq D \quad \forall e{\in}E.\ (\nu_e \in \{+,\circ\} \Rightarrow \Gamma \vdash A_e \leqslant B_e)\\ \wedge\ (\nu_e \in \{-,\circ\} \Rightarrow \Gamma \vdash B_e \leqslant A_e)\end{array}}{\Gamma \vdash [\mathrm{m}_d :_{\nu_d} A_d]_{d\in D} \leqslant [\mathrm{m}_e :_{\nu_e} B_e]_{e\in E}}$$

$$(\textsc{SubObjVar})\ \frac{\forall d \in D.\ \nu_d = \circ\ \vee\ \nu_d = \nu'_d}{\Gamma \vdash [\mathrm{m}_d :_{\nu_d} A_d]_{d\in D} \leqslant [\mathrm{m}_d :_{\nu'_d} A_d]_{d\in D}}$$

$$(\textsc{SubRec})\ \frac{\Gamma, Y{\leqslant}\mathit{Top}, X{\leqslant}Y \vdash A \leqslant B}{\Gamma \vdash \mu(X)A \leqslant \mu(Y)B}$$

$$(\textsc{SubUniv})\ \frac{\Gamma \vdash A' \leqslant A \quad \Gamma, X{\leqslant}A' \vdash B \leqslant B'}{\Gamma \vdash \forall(X{\leqslant}A)B \leqslant \forall(X{\leqslant}A')B'}$$

$$(\textsc{SubExist})\ \frac{\Gamma \vdash A \leqslant A' \quad \Gamma, X{\leqslant}A \vdash B \leqslant B'}{\Gamma \vdash \exists(X{\leqslant}A)B \leqslant \exists(X{\leqslant}A')B'}$$

Figure 4. Subtyping

following semantic domains for values and heaps, respectively:

$$(1)\qquad \begin{aligned} D_{\mathit{Val}} &= (D_{\mathit{Heaps}} \times D_{\mathit{Val}} \rightharpoonup D_{\mathit{Heaps}} \times D_{\mathit{Val}}) + \ldots \\ D_{\mathit{Heaps}} &= \mathit{Loc} \rightharpoonup_{\mathit{fin}} D_{\mathit{Val}} \end{aligned}$$

A simple cardinality argument shows that there are no set-theoretic solutions (*i.e.* where $A \rightharpoonup B$ denotes the set of all partial functions from A to B) satisfying the equations in (1). A possible solution is to use a domain-theoretic approach, as done for imperative objects in [23, 31].

However, in a model of a typed calculus one wants even more: naively taking a collection *Type* of subsets $\tau \subseteq D_{\mathit{Val}}$ as interpretations of syntactic types does not work, since values generally depend on the heap and a typed model should guarantee that all heap access operations are type-correct. One is led to (i) also consider *heap typings*: partial maps $\Psi \in \mathit{HeapTypings} = \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Type}$ that map heap locations to the set of possible values that may be stored, and (ii) to refine the collection of types to take heap typings into account: a type will then consist of values paired with heap typings which describe the necessary requirements on heaps. Thus (i) and (ii) suggest to take

$$(2)\qquad \begin{aligned} \mathit{Types} &= \mathcal{P}(\mathit{HeapTypings} \times D_{\mathit{Val}}) \\ \mathit{HeapTypings} &= \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Types} \end{aligned}$$

Again, a cardinality argument shows the impossibility of defining these sets.

A final obstacle to modelling imperative languages, albeit independent of the higher-order nature of heaps, is due to dynamic allocation in the heap. It results in heap typings that may *vary* in the course of a computation, reflecting the changing 'shape' of the

$\boxed{\Gamma \vdash a : A}$

$$(\textsc{Sub})\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash a : B}$$

$$(\textsc{Var})\ \frac{\Gamma_1, x{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, x{:}A, \Gamma_2 \vdash x : A}$$

Procedure types

$$(\textsc{Lam})\ \frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A)b : A \to B}$$

$$(\textsc{App})\ \frac{\Gamma \vdash a : B \to A \quad \Gamma \vdash b : B}{\Gamma \vdash a\ b : A}$$

Object types (where $A \equiv [\mathrm{m}_d :_{\nu_d} A_d]_{d\in D}$)

$$(\textsc{Obj})\ \frac{\forall d{\in}D.\ \Gamma, x_d{:}A \vdash b_d : A_d}{\Gamma \vdash [\mathrm{m}_d{=}\varsigma(x_d{:}A)b_d]_{d\in D} : A}$$

$$(\textsc{Inv})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \nu_e \in \{+,\circ\}}{\Gamma \vdash a.\mathrm{m}_e : A_e}$$

$$(\textsc{Upd})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \Gamma, x{:}A \vdash b : A_e \quad \nu_e \in \{-,\circ\}}{\Gamma \vdash a.\mathrm{m}_e \coloneqq \varsigma(x{:}A)b : A}$$

$$(\textsc{Clone})\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{clone}\ a : A}$$

Recursive types

$$(\textsc{Unfold})\ \frac{\Gamma \vdash a : \mu(X)A}{\Gamma \vdash \text{unfold}_{\mu(X)A}\ a : \{\!\!\{X \mapsto \mu(X)A\}\!\!\}(A)}$$

$$(\textsc{Fold})\ \frac{\Gamma \vdash a : \{\!\!\{X \mapsto \mu(X)A\}\!\!\}(A)}{\Gamma \vdash \text{fold}_{\mu(X)A}\ a : \mu(X)A}$$

Bounded quantified types

$$(\textsc{TAbs})\ \frac{\Gamma, X{\leqslant}A \vdash b : B}{\Gamma \vdash \Lambda(X{\leqslant}A)b : \forall(X{\leqslant}A)B}$$

$$(\textsc{TApp})\ \frac{\Gamma \vdash a : \forall(X{\leqslant}A)B \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash a[A'] : \{\!\!\{X \mapsto A'\}\!\!\}(B)}$$

$$(\textsc{Pack})\ \frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash \{\!\!\{X \mapsto C\}\!\!\}(a) : \{\!\!\{X \mapsto C\}\!\!\}(B)}{\Gamma \vdash (\text{pack}\ X{\leqslant}A = C\ \text{in}\ a{:}B) : \exists(X{\leqslant}A)B}$$

$$(\textsc{Open})\ \frac{\Gamma \vdash a : \exists(X{\leqslant}A)B \quad \Gamma \vdash C \quad \Gamma, X{\leqslant}A, x{:}B \vdash b : C}{\Gamma \vdash (\text{open}\ a\ \text{as}\ X{\leqslant}A, x{:}B\ \text{in}\ b{:}C) : C}$$

Figure 5. Typing of terms

heap. However, as is the case for many high-level languages, the object calculus is well-behaved in this respect:

- inside the language, there is no possibility of deallocating heap locations; and
- only weak (*i.e.* type-preserving) updates are allowed.

As a consequence, *extensions* are the only changes of heap typings that need to be considered. Intuitively, values that rely on heaps with typing Ψ will also be type-correct for extended heaps, with an extended heap typing $\Psi' \sqsupseteq \Psi$. For this reason, semantic models of dynamic allocation typically lend themselves to a Kripke-style presentation, where all semantic entities are indexed by *possible worlds* drawn from the set of heap typings, partially (pre-) ordered by heap extension.

Rather than trying to extend the already complex domain-theoretic models to heap typings and dynamic allocation, we will use the step-indexing technique. Being based directly on the operational semantics, this provides an alternative that has less mathematical overhead. In particular, there is no need to find semantic domains satisfying (1); we can simply have D_{Val} be the set of closed values and use syntactic procedures in place of set-theoretic functions. Moreover, it is relatively easy to also model impredicative second-order types in the step-indexed model of [5, 8], which is crucial for the interpretation of object types we develop below. This is non-trivial in the domain-theoretic models, at best.

The circularity in (2) is resolved by considering a stratification based on a notion of 'k-step execution safety', in contrast to the information ordering employed in domain theory: the central idea is that a term has a type τ with approximation k if this assumption cannot be proved wrong (in the sense of reaching a stuck state) in any context by executing fewer than k steps. The key insight with respect to constructing sets (2) is that all language constructs operating on the heap (reading, updating, and allocating) each consume one step. Thus, in order to determine whether a heap typing-value pair $\langle \Psi, v\rangle$ belongs to a type τ with approximation k it is sufficient to know the types of the stored values on which v relies (as recorded by Ψ) only up to level $k-1$. The true meaning of types and heap typings is then obtained by taking the limit over all approximations.

The preceding considerations are now formalized, building on the model originally developed by Ahmed *et al.* for an ML-like language with general references and impredicative second-order types [5, 7, 8]. Apart from some notational differences, the definitions in Section 3.1 are the same as in [5]. Section 3.2 adds subtyping, while Section 3.3 deals with procedure types and Section 3.4 revisits reference types. The semantics of object types is presented in Section 3.5, and constitutes the main contribution of this paper. We further deviate from [5] by adding bounds to the second-order types in Section 3.6, and by using iso-recursive instead of equi-recursive types in Section 3.7.

3.1 The Semantic Model

To make the (circular) definition of types and heap typings from (2) work, the step-indexed semantics considers triples with an additional natural numbers index, rather than just pairs. First, we inductively define two families $(PreType_k)_{k\in\mathbb{N}}$ of *pre-types*, and $(HeapTyping_k)_{k\in\mathbb{N}}$ of *heap pre-typings*, by

$$\begin{aligned}\tau \in PreType_0 &\;\Leftrightarrow\; \tau = \emptyset\\ \tau \in PreType_{k+1} &\;\Leftrightarrow\; \tau \in \mathcal{P}(\mathbb{N} \times (\textstyle\bigcup_{j\le k} HeapTyping_j) \times CVal)\\ &\qquad\qquad \wedge\; \forall\langle j, \Psi, v\rangle \in \tau.\; j \le k \;\wedge\; \Psi \in HeapTyping_j\end{aligned}$$

where $HeapTyping_k = Loc \rightharpoonup_{fin} PreType_k$. Clearly $PreType_k \subseteq PreType_{k+1}$ and thus $HeapTyping_k \subseteq HeapTyping_{k+1}$ for all k.

Now it is possible to set

$$\begin{aligned}\tau \in PreType \;\Leftrightarrow\; &\tau \in \mathcal{P}(\mathbb{N} \times (\textstyle\bigcup_j HeapTyping_j) \times CVal)\\ &\wedge\; \forall\langle j, \Psi, v\rangle \in \tau.\; \Psi \in HeapTyping_j\end{aligned}$$

When writing $\langle k, \Psi, v\rangle$ in the following we always implicitly assume that $\Psi \in HeapTyping_k$.

Definition 3.1 (Semantic approximation)**.** For any pre-type τ we call $\lfloor\tau\rfloor_k$ the *k-th approximation of* τ and define it as the subset containing all elements of τ that have an index strictly less than k:

$$\lfloor\tau\rfloor_k = \{\langle j, \Psi, v\rangle \in \tau \mid j < k\}$$

This is lifted pointwise to (partial) functions to pre-types:

$$\lfloor\Psi\rfloor_k = \lambda l \in dom(\Psi).\; \lfloor\Psi(l)\rfloor_k$$

From these definitions we have:

Proposition 3.2 (Stratification)**.** *For all $\tau \in PreType$ and $k \in \mathbb{N}$,*

$$\lfloor\tau\rfloor_k \in PreType_k.$$

So in particular, if $\langle k, \Psi, v\rangle \in \tau$ and $l \in dom(\Psi)$ then $\Psi(l) \in PreType_j$ for some $j \le k$. This is captured by the following 'stratification invariant', which all the constructions on (pre-) types will satisfy, and which ensures the well-foundedness of the whole construction:

Stratification invariant. For all pre-types τ, $\lfloor\tau\rfloor_{k+1}$ cannot depend on any pre-type beyond approximation k.

As indicated above, in order to take dynamic allocation into account we consider a possible worlds model. Intuitively we think of a pair (k, Ψ) as describing the *state* of a heap h, where Ψ lists locations in h that are guaranteed to be allocated, and contains the approximate type (up to approximation k) of the stored values. In the course of a computation, there are three different situations where the heap state changes:

- New objects are allocated on the heap, which is reflected by a heap pre-typing Ψ' with additional locations compared to Ψ. This operation does not affect any of the previously stored objects, so Ψ' will be an extension of Ψ.
- The program executes for $k-j$ steps, for some $j \le k$, without accessing the heap. This is reflected by a heap state $(j, \lfloor\Psi\rfloor_j)$ that 'forgets' that we have a more precise approximation, and guarantees that the heap is safe only for j execution steps.
- The heap is updated, in such a way that all typing guarantees of Ψ are preserved. Thus updates will be reflected by an information forgetting extension, as in the previous case. However, because of the step taken by the update itself, in this case we necessarily have that $j < k$.

The following definition of state extension captures these possible evolutions of a state.

Definition 3.3 (State extension)**.** The *state extension* $\sqsubseteq$ is the relation on $\mathbb{N} \times (Loc \rightharpoonup_{fin} PreType)$ defined by

$$\begin{aligned}(k, \Psi) \sqsubseteq (j, \Psi') \;\Leftrightarrow\; &j \le k \;\wedge\; dom(\Psi) \subseteq dom(\Psi')\\ &\wedge\; \forall l \in dom(\Psi).\; \lfloor\Psi'\rfloor_j(l) = \lfloor\Psi\rfloor_j(l)\end{aligned}$$

The step-indexing technique relies on the approximation of the 'true' set of values that constitute a type, by all those values which behave accordingly unless a certain number of computation steps is taken. Reducing the number of available steps, we will only be able to make less distinctions. Moreover, if for instance a procedure relies on locations in a heap as described by a state (k, Ψ), we can safely apply it after further allocations. In fact, if we are only interested in safely executing the procedure for

$j < k$ steps, a correspondingly approximate heap will suffice. These conditions are captured precisely by state extension, so we require our semantic types to be closed under state extension:

Definition 3.4 (Semantic types and heap typings)**.** The set *Type* of *semantic types* is the subset of *PreType* defined by

$$\tau \in \mathit{Type} \;\Leftrightarrow\; \forall k, j \geq 0.\ \forall \Psi, \Psi'.\ \forall v \in \mathit{CVal}.\ (k, \Psi) \sqsubseteq (j, \Psi') \wedge \langle k, \Psi, v\rangle \in \tau \Rightarrow \langle j, \Psi', v\rangle \in \tau$$

We define the set $\mathit{HeapTyping} = \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Type}$ of *heap typings*, ranged over by Ψ in the following, as the subset of heap pre-typings that map to semantic types.

As explained by Ahmed in [5], this structure may be viewed as an instance of Kripke models of intuitionistic logic where states are the possible worlds, state extension is the reachability relation between worlds, and where the closure under state extensions corresponds to Kripke monotonicity.

Definition 3.5 (Well-typed heap)**.** A heap h *has type* Ψ *with approximation* k, written as $h :_k \Psi$, if $\mathit{dom}(\Psi) \subseteq \mathit{dom}(h)$ and

$$\forall j < k.\ \forall l \in \mathit{dom}(\Psi).\ \langle j, \lfloor \Psi \rfloor_j, h(l)\rangle \in \Psi(l)$$

Semantic types only contain values, but we also need to associate types to terms that are not values. We do this in two steps, first for closed terms, then for arbitrary ones. A closed term has a certain type with approximation k with respect to some heap typing Ψ, if in all heaps that are allowed by Ψ the term behaves like an element of the type for k computation steps. In general, before reducing to a value the term will execute for j steps, and possibly allocate some new heap locations in doing so. The state describing the final heap will therefore be an extension of the state describing the initial store, and it only needs to be safe for the remaining $k-j$ steps. Similarly, the final value needs to be in the original type only for another $k - j$ steps. The next definition makes this precise.

Definition 3.6 (Closed Term $:_{k,\Psi}$ Semantic Type)**.** For a term a we define that a has type τ with respect to the state (k, Ψ) as:

$$\begin{aligned} a :_{k,\Psi} \tau \;\Leftrightarrow\; & \mathit{fv}(a) = \emptyset \wedge \forall j < k, h, h', b. \\ & (h :_k \Psi \wedge \langle h, a\rangle \rightarrow^j \langle h', b\rangle \wedge \langle h', b\rangle \not\rightarrow) \\ & \Rightarrow \exists \Psi'.\ (k, \Psi) \sqsubseteq (k-j, \Psi') \\ & \quad \wedge h' :_{k-j} \Psi' \wedge \langle k-j, \Psi', b\rangle \in \tau \end{aligned}$$

Even though the terms we evaluate are closed, when type-checking their subterms we also have to reason about open terms. Typing open terms is done with respect to a semantic type environment which maps variables to semantic types. We reduce typing open terms to typing closed terms by substituting all free variables with appropriate closed values. This is done by a value environment (a finite map from variables to closed values) that agrees with the type environment.

Definition 3.7 ($\sigma :_{k,\Psi} \Sigma$)**.** We say that *value environment σ agrees with semantic type environment Σ, with respect to the state (k, Ψ)*, if $\forall x \in \mathit{dom}(\Sigma).\ \sigma(x) :_{k,\Psi} \Sigma(x)$. We denote this by $\sigma :_{k,\Psi} \Sigma$.

Definition 3.8 (Semantic typing judgement)**.** We say that a term a (possibly with free variables, but not containing locations), *has type* τ with respect to a semantic type environment Σ, written as $\Sigma \models a : \tau$, if after substituting well-typed values for the free variables of a, we obtain a closed term that has type τ for any number of computation steps. More precisely:

$$\begin{aligned} \Sigma \models a : \alpha \;\Leftrightarrow\; & \mathit{fv}(a) \subseteq \mathit{dom}(\Sigma) \\ & \wedge \forall k \geq 0.\ \forall \Psi.\ \forall \sigma :_{k,\Psi} \Sigma.\ \sigma(a) :_{k,\Psi} \alpha \end{aligned}$$

By construction, the semantic typing judgment enforces that all terms that are typable with respect to it do not produce type errors when evaluated.

Definition 3.9 (Safe for k steps)**.** We call a configuration $\langle h, a\rangle$ *safe for k steps*, if the term a does not get stuck in less than k steps when evaluated in the heap h, *i.e.* we define the set of all such configurations by

$$\begin{aligned} \mathsf{Safe}_k = \{\langle h, a\rangle \mid \forall j < k.\ \forall h', b.\ & \langle h, a\rangle \rightarrow^j \langle h', b\rangle \\ & \wedge \langle h', b\rangle \not\rightarrow \;\Rightarrow\; b \in \mathit{Val}\} \end{aligned}$$

Definition 3.10 (Safety)**.** We call a configuration *safe* if it does not get stuck in any number of steps.

Theorem 3.11 (Safety)**.** *For all programs a, if $\emptyset \models a : \alpha$, then for all heaps h we have that $\langle h, a\rangle \in \mathsf{Safe}$.*

Proof sketch. One first easily shows that, if $a :_{k,\Psi} \tau$ and $h :_k \Psi$, then $\langle h, a\rangle \in \mathsf{Safe}_k$. The theorem then follows by observing that any h has the empty heap type, with any approximation k. □

This result is much more direct than in a subject-reduction proof [35]. However, unlike with subject-reduction, the validity of the typing rules still needs to be proved with respect to the model. We do this in two steps. In the remainder of this section we introduce the specific semantic types of our model, and prove that they satisfy certain semantic typing lemmas. These proofs are similar in spirit to proving the 'fundamental theorem' of Kripke logical relations [25]. Then, in Section 4 we prove the soundness of the rules of the initial type system with respect to these typing lemmas.

Even though the semantic typing lemmas are constructed so that they directly correspond to the rules of the original type system, there is a big difference between the two. While the semantic typing lemmas allow us to logically derive valid semantic judgments using other valid judgments as premises, the typing rules are just syntax which is used in the inductive definitions of the typing and subtyping relations.

3.2 Subtyping

Since types in the step-indexed model are sets (with some additional properties), the natural subtyping relation is set inclusion. This subtyping relation forms a complete lattice on semantic types, where infima and suprema are given by set-theoretic intersections and unions, respectively. The least element is $\bot = \emptyset$, while the greatest is

$$\top = \{\langle j, \Psi, v\rangle \mid j \in \mathbb{N}, \Psi \in \mathit{HeapTyping}_j, v \in \mathit{CVal}\}.$$

Obviously $\bot$ and $\top$ satisfy both the stratification invariant (*i.e.* they are pre-types) and the closure under state extension condition, so they are indeed semantic types.

We can easily show the standard subsumption property

Lemma 3.12 (Subsumption)**.** *If $\Sigma \models a : \alpha$ and $\alpha \subseteq \beta$ then $\Sigma \models a : \beta$.*

While it is very easy to define subtyping this way, the interaction between subtyping and the other features of the type system, in particular the object types, is far from trivial.

3.3 Procedure Types

Intuitively, a procedure has type $\alpha \rightarrow \beta$ if, when invoked with an argument of type α, it produces a result of type β. In a step-indexed model, a procedure has type $\alpha \rightarrow \beta$ for k computation steps if when applied to any well-typed argument of type α it produces a result that has type β for another $k-1$ steps. This is because the procedure application itself takes one computation step, and the only way to use a procedure is by applying it to some argument.

(SemLam) $\Sigma[x := \alpha] \models b : \beta \implies \Sigma \models \lambda x.\, b : \alpha \to \beta$

(SemApp) $(\Sigma \models a : \beta \to \alpha \;\wedge\; \Sigma \models b : \beta \implies \Sigma \models a\, b : \alpha$

(SemSubProc) $\alpha' \subseteq \alpha \;\wedge\; \beta \subseteq \beta' \implies \alpha \to \beta \subseteq \alpha' \to \beta'$

Figure 6. Typing lemmas: procedure types

Additionally, we have to take into account that the procedure can also be applied after some computation steps that extend the heap. So, for every $j < k$ and for every heap typing Ψ' such that $(k, \Psi) \sqsubseteq (j, \Psi')$, when applying the procedure to a value in type α for j steps with respect to Ψ', the result must have type β for j steps with respect to Ψ'. This fits nicely with the possible worlds reading of procedure types as intuitionistic implication.

Definition 3.13 (Procedure types)**.** If α and β are semantic types, then $\alpha \to \beta$ consists of those triples $\langle k, \Psi, \lambda x.\, b\rangle$ such that for all $j < k$, heap typings Ψ' and closed values v:

$$((k, \Psi) \sqsubseteq (j, \Psi') \wedge \langle j, \Psi', v\rangle \in \alpha) \;\Rightarrow\; \{\!|x \mapsto v|\!\}(b) :_{j,\Psi'} \beta$$

Proposition 3.14. *If α and β are semantic types, then $\alpha \to \beta$ is also a semantic type.*

Figure 6 contains the semantic typing lemmas associated with procedure types. The procedure type constructor is of course contravariant in the argument type and covariant in the result type.

Lemma 3.15 (Procedure types)**.** *The three semantic typing lemmas shown in Figure 6 are valid implications.*

Proof sketch. The validity of (SemApp) and (SemLam) is proved in [5]. Verifying (SemSubProc) is simply a matter of unfolding the definitions. □

3.4 Revisiting Reference Types

While our calculus does not have references syntactically, we will use the model of references from [5, 8] in our construction underlying object types. In order to interpret the variance annotations in object types, we also need to introduce readable reference types and writable reference types, with covariant and contravariant subtyping, respectively [27, 32].

A heap typing associates to each allocated location the precise type that can be used when reading from it and writing to it. So all heap locations support both reading and writing at a certain type, and we do not have read-only or write-only locations. Intuitively, for the readable reference types and the writable ones the precise type of the locations is only partially known, so that without additional information only one of the two operations is safe at a meaningful type.

We first recall the definition of reference types from [5, 8].

Definition 3.16 (Reference types)**.** If τ is a semantic type then

$$\text{ref}_\circ\tau = \{\langle k, \Psi, l\rangle \mid \lfloor\Psi(l)\rfloor_k = \lfloor\tau\rfloor_k\}$$

According to this definition, a location l has type $\text{ref}_\circ\tau$ if the type associated to l by the heap typing Ψ is approximately τ. Semantic approximation is used to satisfy the stratification invariant, and is operationally justified by the fact that reading from a location or writing to it takes one computation step. So, l has type $\text{ref}_\circ\tau$ for k steps if all values that are read from l or written to l have type τ for $k - 1$ steps.

The readable reference type $\text{ref}_+\tau$ is similar to $\text{ref}_\circ\tau$, but poses less constraints on the heap typing Ψ: it only requires that $\Psi(l)$ is a subtype of τ, as before up to some approximation.

Definition 3.17 (Readable reference types)**.** If τ is a type then

$$\text{ref}_+\tau = \{\langle k, \Psi, l\rangle \mid \lfloor\Psi\rfloor_k(l) \subseteq \lfloor\tau\rfloor_k\}$$

The value stored at location l also has type τ by subsumption, and therefore can be read and safely used as a value of type τ. However, the true type of location l is in general unknown, so writing any value to it could be unsafe (the true type of l might be the empty type $\bot$). On the other hand, knowing that a location has type $\text{ref}_+\tau$ does not mean that we cannot write to it: it simply means that we do not know the type of the values that can be written to it, so in the absence of further information no writing can be guaranteed to be type safe[2].

Dually, the type $\text{ref}_-\tau$ of writable references contains all those locations l whose type associated by Ψ is a supertype of τ.

Definition 3.18 (Writable reference types)**.** If τ is a type then

$$\text{ref}_-\tau = \{\langle k, \Psi, l\rangle \mid \lfloor\tau\rfloor_k \subseteq \lfloor\Psi\rfloor_k(l)\}$$

We can safely write a value of type τ to a location of type $\text{ref}_-\tau$, since this value also has the real type of location l by subsumption. However, the real type of such locations can be arbitrarily general. In particular it can be $\top$, the type of all values. Thus a location about which we only know that it has type $\text{ref}_-\tau$ can only be read safely at type $\top$.

With these definitions, the usual reference type from [5, 8] can be recovered as the intersection of a readable and a writable reference type:

$$\text{ref}_\circ\tau = \text{ref}_+\tau \cap \text{ref}_-\tau$$

Hence $\text{ref}_+\tau$ and $\text{ref}_-\tau$ are both supertypes of $\text{ref}_\circ\tau$. It can also be easily shown that the readable reference type constructor is covariant, while the writable reference one is contravariant. The usual reference types are therefore obviously invariant. For a variance annotation $\nu \in \{\circ, +, -\}$ we use ref_ν to stand for the reference type constructor with this variance. Note that, strictly speaking, the set $\text{ref}_\nu\tau$ is not a semantic type since for our calculus locations are not values. The definition of object types we give below does not rely on $\text{ref}_\nu\tau$ being a type.

3.5 Object Types

Giving a semantics to object types is challenging. Not surprisingly, the definition of object types is more complex than the other definitions of types encountered in this paper. A look at the typing rules from Section 2 gives an indication why this is the case. First, an adequate interpretation of object types must permit subtyping both in width and in depth, taking the variance annotations into account accordingly. Second, in contrast to all the other types we consider which have just a single elimination rule, once constructed, objects support three different operations: invocation, update, and cloning. The definition of object types must ensure the consistent use of an object through all possible future operations. That is, all the requirements on which invocation, update or cloning rely must already be established at object creation time.

We will now explain the construction in detail. It is well-known that invocation, update, cloning and the desired subtyping properties impose conflicting requirements on typed objects:

- Our decision to store methods in the heap as procedures, together with the 'self-application' operational semantics of method invocation (Red-Inv in Figure 3), suggests that object types are somewhat similar to recursive types of records of references holding procedures that take the enclosing record as argument:

$$[\text{m}_d : \tau_d]_{d\in D} \approx \mu(\alpha).\{\text{m}_d : \text{ref}_\circ(\alpha \to \tau_d)\}_{d\in D}.$$

[2] Note that this is conceptually different from the immutable reference types modeled in [5] using singleton types.

Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$.

(SEMOBJ)	$(\forall d \in D.\ \Sigma[x_d := \alpha] \models b_d : \tau_d)$	$\Longrightarrow$	$\Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$
(SEMINV)	$(\Sigma \models a : \alpha\ \wedge\ e \in D\ \wedge\ \nu_e \in \{+,\circ\})$	$\Longrightarrow$	$\Sigma \models a.\mathrm{m}_e : \tau_e$
(SEMUPD)	$(\Sigma \models a : \alpha\ \wedge\ e \in D\ \wedge\ \nu_e \in \{-,\circ\}\ \wedge\ \Sigma[x := \alpha] \models b : \tau_e)$	$\Longrightarrow$	$\Sigma \models a.\mathrm{m}_e \coloneqq \varsigma(x)b : \alpha$
(SEMCLONE)	$\Sigma \models a : \alpha$	$\Longrightarrow$	$\Sigma \models \mathrm{clone}\ a : \alpha$
(SEMSUBOBJ)	$(D \subseteq E\ \wedge\ (\forall d \in D.\ \nu_d \in \{+,\circ\} \Rightarrow \alpha_d \subseteq \beta_d)$ $\wedge\ (\forall d \in D.\ \nu_d \in \{-,\circ\} \Rightarrow \beta_d \subseteq \alpha_d))$	$\Longrightarrow$	$[\mathrm{m}_e :_{\nu_e} \alpha_e]_{e\in E} \subseteq [\mathrm{m}_d :_{\nu_d} \beta_d]_{d\in D}$
(SEMSUBOBJVAR)	$(\forall d \in D.\ \nu_d = \circ\ \vee\ \nu_d = \nu'_d)$	$\Longrightarrow$	$[\mathrm{m}_d :_{\nu_d} \alpha_d]_{d\in D} \subseteq [\mathrm{m}_d :_{\nu'_d} \alpha_d]_{d\in D}$

Figure 7. Typing lemmas: object types

However, the invariance of the reference type constructor blocks any form of subtyping, even just in width[3].

- A combination of type recursion and an existential quantifier that uses the recursion variable as bound would allow us to enforce covariance for the positions of the recursion variable, and thus have subtyping in width:

  $$[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \approx \mu(\alpha).\exists\alpha' \subseteq \alpha.\{\mathrm{m}_d : \mathrm{ref}_\circ(\alpha' \to \tau_d)\}_{d\in D}$$

  This is similar to the encoding of self types explored in [2, 3].
- For subtyping in depth with respect to the variance annotations we simply use the readable and writable reference types defined in the previous section:

  $$[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \approx \mu(\alpha).\exists\alpha'{\subseteq}\alpha.\{\mathrm{m}_d : \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)\}_{d\in D}.$$

  Still, by keeping α' abstract, neither the typing rule for method invocation (INV in Figure 5), nor the one for object cloning (CLONE) is validated.
- By explicitly enforcing in the definition of object types that the object value itself belongs in fact to this existentially quantified α', the assumptions become sufficiently strong to repair the invocation case. In fact, by enforcing this not only for the current object value, but also for all 'very similar' values, maybe not even created yet, the case of cloning is also covered.

The following definition formalizes this construction.

Definition 3.19 (Object types)**.** Let $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$, defined as the set of all triples $\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle$ such that $D \subseteq E$ and

(1) $\quad \exists\alpha'.\ \alpha' \in \mathit{Type}\ \wedge\ \alpha' \subseteq \lfloor\alpha\rfloor_k$

(2) $\quad \wedge\ (\forall d \in D.\ \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$

(3) $\quad \wedge\ (\forall j < k.\ \forall\Psi'.\ \forall \{\mathrm{m}_e{=}l'_e\}_{e\in E}.\ (k, \Psi) \sqsubseteq (j, \Psi')$
$\quad\quad \wedge\ (\forall e \in E.\ \lfloor\Psi'\rfloor_j (l'_e) = \lfloor\Psi\rfloor_j (l_e))$
$\quad\quad \Rightarrow \langle j, \lfloor\Psi'\rfloor_j, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha')$

The condition stating that $D \subseteq E$ ensures that all values in an object type provide *at least* the required methods listed by this type, but can also provide more. Clearly this is necessary for subtyping in width. Condition (1) postulates the existence of a more specific type α', which may be thought of as the 'true' type of the object $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ (up to approximation k). The subsequent conditions are then all stated in terms of α' rather than α. Condition (2) states the requirements for the methods in terms of the reference type constructors introduced in Section 3.4.

As explained above, in order to be able to invoke methods we must know that $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ belongs to the more specific type α' for all $j < k$ steps (which suffices since application consumes a step). In the particular case where Ψ' is Ψ and $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ is $\{\mathrm{m}_e{=}l_e\}_{e\in E}$ condition (3) states exactly this. We need the more general formulation in order to ensure that the clones of the considered object also belong to the same type α'. Therefore we enforce that no matter how an object value $\{\mathrm{m}_e{=}l'_e\}_{e\in E}$ is constructed it belongs to type α', provided that it satisfies the same typing assumptions as $\{\mathrm{m}_e{=}l_e\}_{e\in E}$, with respect to a possibly extended heap typing Ψ'. Allowing for state extension is necessary since cloning itself allocates new locations not present in the original Ψ, and also because cloning can be performed after some intermediate computation steps that result in further allocations.

We show that this definition of object types actually makes sense, in that it defines a semantic type. This is not immediately obvious because of the recursion.

Proposition 3.20. *If $\tau_d \in \mathit{Type}$ for all $d \in D$, then we also have that $[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D} \in \mathit{Type}$.*

Proof sketch. To show that $[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ is uniquely determined, one could prove that (1)–(3) determine a *contractive* map *Type*→*Type* and then use the general results about recursive types from [10]. Alternatively, we observe that $\tau = \bigcup_k \lfloor\tau\rfloor_k$ for all types τ, and then directly argue that Definition 3.19 defines $\lfloor[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}\rfloor_k$ only in terms of $\lfloor[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}\rfloor_j$ for $j < k$. That $[\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ is closed under state extension, relies on the transitivity of state extension and is easily verified. ☐

Figure 7 presents the semantic typing lemmas for object types and their subtyping.

Lemma 3.21 (Object types)**.** *All the semantic typing lemmas shown in Figure 7 are valid implications.*

Proof sketch. The semantic typing lemmas are proved independently. We sketch this for (SEMOBJ). For $\alpha = [\mathrm{m}_d :_{\nu_d} \tau_d]_{d\in D}$ and assuming $\Sigma[x_d := \alpha] \models b_d : \tau_d$ for all $d \in D$, we must show that $\Sigma \models [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$.

So let $k \geq 0$, σ and Ψ be such that $\sigma :_{k,\Psi} \Sigma$. By Definition 3.8 (Semantic typing judgement) we must prove that $\sigma([\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D}) :_{k,\Psi} \alpha$, or equivalently (after suitable α-renaming), that $[\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d\in D} :_{k,\Psi} \alpha$ holds. Now let h, h' and b' be such that $h :_k \Psi$ and

$$\langle h, [\mathrm{m}_d{=}\varsigma(x_d)\sigma(b_d)]_{d\in D}\rangle \to^j \langle h', b'\rangle$$

[3] It turns out that even without the reference types (*e.g.* for the functional object calculus) the contravariance of the procedure type constructor in its first argument would still cause any sort of subtyping to fail.

For all non-expansive $F, G : \mathit{Type} \to \mathit{Type}$,

(SEMTABS)	$(\forall \tau \in \mathit{Type}.\ \tau \subseteq \alpha \Rightarrow \Sigma \models a : F(\tau))$	$\Longrightarrow$	$\Sigma \models \Lambda.\, a : \forall_\alpha F$
(SEMTAPP)	$(\Sigma \models a : \forall_\alpha F\ \wedge\ \tau \in \mathit{Type}\ \wedge\ \tau \subseteq \alpha)$	$\Longrightarrow$	$\Sigma \models a[] : F(\tau)$
(SEMPACK)	$(\exists \tau \in \mathit{Type}.\ \tau \subseteq \alpha\ \wedge\ \Sigma \models a : F(\tau))$	$\Longrightarrow$	$\Sigma \models \text{pack } a : \exists_\alpha F$
(SEMOPEN)	$(\Sigma \models a : \exists_\alpha F\ \wedge\ \forall \tau {\in} \mathit{Type}.\ \tau \subseteq \alpha \Rightarrow \Sigma[x := F(\tau)] \models b : \beta)$	$\Longrightarrow$	$\Sigma \models \text{open } a \text{ as } x \text{ in } b : \beta$
(SEMSUBUNIV)	$(\beta \subseteq \alpha\ \wedge\ \forall \tau \in \mathit{Type}.\ \tau \subseteq \beta \Rightarrow F(\tau) \subseteq G(\tau))$	$\Longrightarrow$	$\forall_\alpha F \subseteq \forall_\beta G$
(SEMSUBEXIST)	$(\alpha \subseteq \beta\ \wedge\ \forall \tau \in \mathit{Type}.\ \tau \subseteq \alpha \Rightarrow F(\tau) \subseteq G(\tau))$	$\Longrightarrow$	$\exists_\alpha F \subseteq \exists_\beta G$

Figure 8. Typing lemmas: bounded quantified types

for some $j < k$, and assume that $\langle h', b' \rangle$ is irreducible. From the operational semantics it is clear that $j = 1$ and $b' \equiv \{\mathrm{m}_d{=}l_d\}_{d \in D}$ and that, for some $l_d \notin \mathit{dom}(h)$,

$$h' = h\,[l_d := \lambda(x_d)\sigma(b_d)]_{d \in D}$$

Choosing $\Psi' = \lfloor \Psi\,[l_d := (\alpha \to \tau_d)]_{d \in D} \rfloor_{k-1}$ it is easily seen, that $(k, \Psi) \sqsubseteq (k-1, \Psi')$. Furthermore, from the hypothesis by (SEMLAM) we have that $\Sigma \models \lambda(x_d) b_d : \alpha \to \tau_d$ for all $d \in D$. From this and the assumption that $h :_k \Psi$ it follows that $h' :_{k-1} \Psi'$.

By Definition 3.6 it remains to establish that $\langle k{-}1, \Psi', b' \rangle \in \alpha$. This is achieved by proving by induction on $j \geq 0$ the following more general claim:

Claim 3.22. For all $j \in \mathbb{N}$, Ψ^* and $\{\mathrm{m}_d{=}l_d^*\}_{d \in D}$ we have that

$$\begin{aligned}(k-1, \Psi') \sqsubseteq (j, \Psi^*)\ \wedge\ (\forall d \in D.\ \lfloor \Psi^* \rfloor_j (l_d^*) = \lfloor \Psi' \rfloor_j (l_d))\\ \Rightarrow\ \langle j, \lfloor \Psi^* \rfloor_j, \{\mathrm{m}_d{=}l_d^*\}_{d \in D} \rangle \in \alpha\end{aligned}$$

The key step is in *choosing* α' equal to $\lfloor \alpha \rfloor_j$, and then verifying the conditions (1)–(3) of Definition 3.19 (Object types) with the help of the inductive hypothesis.

The proofs of the other typing lemmas are more straightforward, even though quite lengthy. For (SEMINV), (SEMUPD) and (SEMCLONE), Proposition 2.1 (Unique decomposition) is used to identify reduction sequences corresponding to the evaluation of the subterm a. ☐

Our attempts to prove $\langle k{-}1, \Psi', b' \rangle \in \alpha$ directly have failed, and the generalization to Claim 3.22 seems crucial. In fact, the induction on the step index j resolves the recursion that is inherent to objects due to the self application semantics of method invocation.

3.6 Bounded Quantified Types

Impredicative quantified types in a step-indexed model were previously studied by Ahmed *et al.* for a lambda-calculus with general references, and we follow their presentation [5, 8]. However, unlike in the work of Ahmed *et al.* our quantifiers have bounds, and we are also studying subtyping. It is important to note that the impredicative second-order types were the reason why a semantic stratification of types was needed in [5], as opposed to a syntactic one based on the nesting of reference types [7].

A type constructor (*i.e.* a function from semantic types to semantic types) F is non-expansive if in order to determine whether a term has type $F(\tau)$ with approximation k, it suffices to know the type τ only to approximation k. As we will later show all our type constructors are non-expansive.

Definition 3.23 (Non-expansiveness)**.** A type constructor F : *Type* $\to$ *Type* is *non-expansive* if for all types τ and for all $k \geq 0$ we have $\lfloor F(\tau) \rfloor_k = \lfloor F(\lfloor \tau \rfloor_k) \rfloor_k$.

The definitions of second-order types require that $\forall$ and $\exists$ are only applied to non-expansive type constructors. Note that the non-expansiveness condition in the following definitions ensures that, in order to determine level k of a universal or existential type, a quantification over types τ in $\mathit{PreType}_k$ suffices. This helps avoid the circularity that is otherwise introduced by the *impredicative* quantification.

Definition 3.24 (Bounded universal types)**.** If F : *Type* $\to$ *Type* is non-expansive and $\alpha \in$ *Type*, then we define $\forall_\alpha F$ by $\langle k, \Psi, \Lambda.\, a \rangle \in \forall_\alpha F$ if and only if

$$\begin{aligned}\forall j, \Psi'.\ \forall \tau.\ (k, \Psi) \sqsubseteq (j, \Psi')\ \wedge\ \lfloor \tau \rfloor_j \in \mathit{Type}\ \wedge\ \lfloor \tau \rfloor_j \subseteq \lfloor \alpha \rfloor_j\\ \Rightarrow \forall i < j.\ a :_{i, \lfloor \Psi' \rfloor_i} F(\tau)\end{aligned}$$

Definition 3.25 (Bounded existential types)**.** For all non-expansive F : *Type* $\to$ *Type* and $\alpha \in$ *Type*, the set $\exists_\alpha F$ is defined by $\langle k, \Psi, \text{pack } v \rangle \in \exists_\alpha F$ if and only if

$$\begin{aligned}\exists \tau.\ \lfloor \tau \rfloor_k \in \mathit{Type}\ \wedge\ \lfloor \tau \rfloor_k \subseteq \lfloor \alpha \rfloor_k\\ \wedge\ \forall j < k.\ \langle j, \lfloor \Psi \rfloor_j, v \rangle \in F(\tau)\end{aligned}$$

Proposition 3.26. *If $\alpha \in$ Type and F : Type $\to$ Type is non-expansive, then $\forall_\alpha F$ and $\exists_\alpha F$ are also types.*

Lemma 3.27 (Bounded quantified types)**.** *All the semantic typing lemmas shown in Figure 8 are valid implications.*

Proof sketch. The first four implications are proved as in [5]; the additional precondition $\tau \subseteq \alpha$ in (SEMTAPP) and (SEMPACK) serves to establish the requirements for the bounds. The two subtyping lemmas (SEMSUBUNIV) and (SEMSUBEXIST) are easily proved by just unfolding the definitions. ☐

3.7 Recursive Types

In contrast to most previous work on step-indexed models, we consider iso-recursive rather than equi-recursive types[4], so folds and unfolds are explicit in our syntax and consume computation steps. This is simpler, and sufficient for our purpose. As a benefit, it suffices to require type constructors to be non-expansive, as opposed to the stronger 'contractiveness' requirement in [10].

Definition 3.28 (Recursive types)**.** Let F : *Type* $\to$ *Type* be a non-expansive function. We define the set μF by

$$\begin{aligned}\langle k, \Psi, \text{fold}\, v \rangle \in \mu F\ \Leftrightarrow\ \forall j < k.\ \forall \Psi'.\ (k, \Psi) \sqsubseteq (j, \Psi')\\ \Rightarrow \langle j, \Psi', v \rangle \in F(\mu F)\end{aligned}$$

Proposition 3.29. *For all non-expansive F : Type $\to$ Type, $\mu F \in$ Type is uniquely defined.*

[4] Still, iso-recursive types have been considered by Ahmed for a step-indexed relational model of a lambda calculus [6].

For all non-expansive $F, G : \mathit{Type} \to \mathit{Type}$,

$$
\begin{array}{ll}
(\textsc{SemUnfold}) & \Sigma \models a : \mu F \implies \Sigma \models \mathrm{unfold}\, a : F(\mu F) \\
(\textsc{SemFold}) & \Sigma \models a : F(\mu F) \implies \Sigma \models \mathrm{fold}\, a : \mu F \\
(\textsc{SemSubRec}) & (\forall \alpha, \beta.\ \alpha \subseteq \beta \Rightarrow \\
& \quad F(\alpha) \subseteq G(\beta)) \implies \mu F \subseteq \mu G
\end{array}
$$

Figure 9. Typing lemmas: recursive types

Proof. The well-definedness follows from the observation that $\lfloor \mu F \rfloor_k$ is defined only in terms of $\lfloor F(\mu F) \rfloor_j$ for $j < k$, which by non-expansiveness of F means that it relies only on $\lfloor \mu F \rfloor_j$. The closure under state extension is immediate from the definition. □

Figure 9 presents the semantic typing lemmas for recursive types.

Lemma 3.30 (Recursive types)**.** *All the semantic typing lemmas shown in Figure 9 are valid implications.*

Proof sketch. The validity of (SemFold) and (SemUnfold) are easy consequences of Definition 3.28. For (SemSubRec), one shows by induction on k that $\lfloor \mu F \rfloor_k \subseteq \lfloor \mu G \rfloor_k$. Both the precondition of (SemSubRec) and the non-expansiveness of F and G are used in this proof. □

Lemma 3.31 (Non-expansiveness)**.** *All the type constructors we consider are non-expansive.*

4. Semantic Soundness

In order to prove that well-typed terms are safe to evaluate we relate the syntactic types to their semantic counterparts, and then use the fact that the semantic typing judgement enforces safety by construction (Theorem 3.11). This approach is standard in denotational semantics.In fact, neither the statements nor the proofs of subtyping (Lemma 4.4) and semantic soundness (Theorem 4.5) mention the step-indices explicitly.

Definition 4.1 (Interpretation of types and typing contexts)**.** Let η be a total function from type variables to semantic types.

1. The interpretation $[\![A]\!]_\eta$ of a type A is given by the structurally recursive meaning function in Figure 10.
2. The interpretation of a well-formed typing context Γ with respect to η is given by the function that maps x to $[\![A]\!]_\eta$, for every $x{:}A \in \Gamma$.

Note that in Figure 10 the type constructors used on the left hand sides of the equations are simply syntax, while those on the right hand sides refer to the corresponding semantic constructions, as defined in the previous section.

Recall that non-expansiveness is a necessary precondition for some of the semantic typing lemmas. In particular, the well-definedness of $[\![A]\!]_\eta$ depends on non-expansiveness, due to the use of μ, $\forall_{(\cdot)}$ and $\exists_{(\cdot)}$ in Figure 10. So we begin by showing that the interpretation of types is a non-expansive map.

Lemma 4.2 (Non-expansiveness)**.** $[\![A]\!]_\eta$ *is non-expansive in* η.

Proof sketch. We show that $\left\lfloor [\![A]\!]_\eta \right\rfloor_k = \left\lfloor [\![A]\!]_{\lfloor \eta \rfloor_k} \right\rfloor_k$ holds by induction on the structure of A, relying on Lemma 3.31 for the non-expansiveness of the semantic type constructions. □

$$
\begin{aligned}
[\![\mathit{Top}]\!]_\eta = \top \quad [\![\mathit{Bot}]\!]_\eta &= \bot \quad [\![X]\!]_\eta = \eta(X) \\
[\![A \to B]\!]_\eta &= [\![A]\!]_\eta \to [\![B]\!]_\eta \\
[\![[\mathrm{m}_d :_{\nu_d} A_d]_{d \in D}]\!]_\eta &= \left[\mathrm{m}_d :_{\nu_d} [\![A_d]\!]_\eta\right]_{d \in D} \\
[\![\mu(X)A]\!]_\eta &= \mu(\lambda\alpha{\in}\mathit{Type}.\ [\![A]\!]_{\eta[X:=\alpha]}) \\
[\![\forall(X{\leqslant}A)B]\!]_\eta &= \forall_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathit{Type}.\ [\![B]\!]_{\eta[X:=\alpha]}) \\
[\![\exists(X{\leqslant}A)B]\!]_\eta &= \exists_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathit{Type}.\ [\![B]\!]_{\eta[X:=\alpha]})
\end{aligned}
$$

Figure 10. Interpretation of types

Definition 4.3 ($\eta \models \Gamma$)**.** Let Γ be a well-formed typing context. We say that η *satisfies* Γ, written as $\eta \models \Gamma$, if $\eta(X) \subseteq [\![A]\!]_\eta$ holds for all $X{\leqslant}A$ appearing in Γ.

We show the soundness of the subtyping relation.

Lemma 4.4 (Soundness of subtyping)**.** *If* $\Gamma \vdash A \leqslant B$ *and* $\eta \models \Gamma$ *then* $[\![A]\!]_\eta \subseteq [\![B]\!]_\eta$.

Proof sketch. By induction on the derivation of $\Gamma \vdash A \leqslant B$ and case analysis on the last applied rule. Each case is immediately reduced to one of the subtyping lemmas from Section 3. □

Finally, we prove the semantic soundness of the syntactic type system with respect to the model.

Theorem 4.5 (Semantic soundness)**.** *Whenever* $\Gamma \vdash a : A$ *and* $\eta \models \Gamma$ *it follows that* $[\![\Gamma]\!]_\eta \models a : [\![A]\!]_\eta$.

Proof sketch. By induction on the derivation of $\Gamma \vdash a : A$ and case analysis on the last rule applied. Each case is easily reduced to one of the semantic typing lemmas from Section 3, using a standard type substitution lemma in some cases. □

By Theorems 4.5 (Semantic soundness) and 3.11 (Safety), we have a proof of safety for the type system from Section 2.3.

Corollary 4.6 (Type safety)**.** *Well-typed terms are safe to evaluate.*

5. Conclusion

We have presented a step-indexed model for Abadi and Cardelli's imperative object calculus, and used it to prove the safety of a type system with object types, recursive and second-order types, as well as subtyping.

5.1 Comparison to Related Work

Domain-theoretic models. Abadi and Cardelli give a semantic model for the functional object calculus in [1, 2]. Their type system is comparable to the one we consider here. Types are interpreted as certain partial equivalence relations over an untyped domain model of the calculus. No indication is given on how to adapt this to the imperative execution model.

Based on earlier work by Kamin and Reddy [23], Reus *et al.* consider domain-theoretic models for the imperative object calculus [30, 31, 33], with the goal of proving soundness for the logic of Abadi and Leino [4]. The higher-order store exhibited by the object calculus makes the semantic domains be defined by mixed-variant recursive equations. The dynamic allocation is then addressed by interpreting specifications of the logic as Kripke relations, indexed by store specifications, which are similar to the heap typings used here.

Building on work by Levy [24], an 'intrinsically' typed model of the object calculus is presented in [33], by solving the domain equations in a suitable category of functors. However, only first-order types are considered.

Compared to the domain-theoretic models, the step-indexed model we presented not only soundly interprets a richer type language, but we found it also easier to work with. Being based on the operational semantics, there is no need for explicit continuity conditions, and the admissibility conditions are replaced by the closure under state extension condition, which is usually very easy to check. All that is needed for the construction of recursive and second-order types are non-expansiveness and the stratification invariant.

It is interesting to see how the object construction rule (OBJ) is proved correct in each case. In [30, 31], it directly corresponds to a recursive predicate, whose well-definedness (*i.e.* existence and uniqueness) must be established. This imposes some further restrictions on the semantic types [28]. In the typed model of [33], object construction is interpreted using a recursively defined function, and correspondingly (OBJ) is proved by fixed point induction. In the step-indexed case, the essence of the proof is a more elementary induction on the index (*cf.* Lemma 3.21).

Semantics of object types. Our main contribution in this paper is the novel interpretation of object types in the step-indexed model. The stratification induced by the step indices permits mixed-variance recursive as well as impredicative, second-order types to be constructed. Both are key ingredients in our interpretation of object types; this would be the case even if we considered a syntactic type system without these features. The use of recursive and existentially quantified types is in line with the type-theoretic work on object encodings, which however has mainly focused on object calculi with a functional semantics [16].

Closest to our work is the encoding of imperative objects into an imperative variant of system $F_{\leqslant\mu}$ with updatable records, proposed in [3]. There, objects are interpreted as records containing references to the procedures that represent the methods. As in our case, these records have a recursive and existentially quantified record type. The difference is that additional record fields are included in order to achieve invocation and cloning, and uninitialized fields are used to construct this recursive record in an 'unsafe' way. Finally, variance annotations for imperative objects are not considered.

Step-indexed models. Step-indexed semantic models were introduced by Appel *et al.* in the context of foundational proof-carrying code [9]. Their goal was to construct more elementary and more modular proofs of type soundness that can be easily checked automatically. They were primarily interested in low-level languages, however they also applied their technique to a pure λ-calculus with recursive types [10]. Later Ahmed *et al.* successfully extended it to general references and impredicative polymorphism [5, 7, 8]. The step-indexed semantic model we present extends the one by Ahmed *et al.* with object types and subtyping. In order to achieve this, we refined the reference types from [5] to readable and writable reference types. These are similar to the reference types in the Forsythe programming language [32] and to the channel types of [17, 27].

Subtyping in a step-indexed semantic model was previously considered by Swadi who studied a typed machine language [34]. Our setup is however much different than the one of Swadi. In particular, the subtle issues concerning the subtyping of object types are original to our work.

The previous work on step-indexing focuses on 'semantic type systems', *i.e.* the semantic typing lemmas are used directly for type-checking programs [8, 12, 9, 10]. However, when one considers more complex type systems with subtyping, recursive types or polymorphism, the semantic typing lemmas no longer directly correspond to the usual syntactic rules. These discrepancies can be fixed, but usually at the cost of more complex models, like the one developed by Swadi to track type variables [12, 34]. In Swadi's model an additional semantic kind system is used to track the contractiveness and non-expansiveness of types with free type variables. We avoid having a more complex model (*e.g.* one that tracks type variables) by considering a standard, *syntactic* type system. We use the semantic typing lemmas only to prove the soundness of this syntactic type system, which is more suitable for type checking programs (in particular it can be made decidable [26]).

Type safety proofs. Abadi and Cardelli use subject-reduction to prove the safety of several type systems very similar to the one considered in this paper [2]. Those purely syntactic proofs are very different from the 'semantic' type safety proof we present (for detailed discussions about the differences see [10, 35]). Constructing a step-indexed model is definitely more challenging than proving progress and preservation. However, for our model we could basically reuse the whole model by Ahmed *et al.* and extend it to suit our needs, even though the calculus we are considering is quite different. So one would expect that once enough general models are constructed (*e.g.* [5, 10, 11]), it will become easier to build new models just by mixing and matching. Assuming the existence of an adequate step-indexed model, the effort needed to prove the semantic typing lemmas using 'pencil-and-paper' is somewhat comparable to the one required for a subject-reduction proof. Since each of the semantic typing lemmas is proved in isolation, the resulting type soundness proof is more modular. However, according to [9, 10] the big advantage of step-indexing should kick in when formalizing the proofs in a proof assistant.

Functional object calculus. Our initial experiments on the current topic were done in the context of the functional object calculus [20]. Even though in the functional setting the semantic model is much simpler, both models satisfy the same semantic typing lemmas. Even more, the syntactic type system we considered for the functional calculus is exactly the same as the one in this paper, so all the results in Section 4 directly apply to the functional object calculus: well-typed terms do not get stuck, no matter whether they are evaluated in a functional or an imperative way. It would not be possible to directly prove such a result using subject-reduction, since in that case the syntactic typing judgment for the imperative calculus would also depend on a heap typing, and thus be different from the judgment for the functional calculus. However, since we are not using subject-reduction, we do not need to type-check partially evaluated terms which contain heap locations.

5.2 Future Work

Machine checking proofs. Step-indexed models were introduced with machine-checkable proofs in mind, so the proofs they induce are elementary and very modular. The proof of type safety we present in the extended version of this paper [21] should therefore be very well-suited for translation to some machine-checkable form. This could be eased to a certain extent by the fact that we extended the model of Ahmed *et al.* for which machine-checkable proofs already exist [5].

Generalizing reference types. In the model described in this paper we generalize the reference types from [5, 8] to readable and writable reference types. It turns out that this can be generalized even further. We can have a reference type constructor that takes two other types as arguments: one that represents the most general type that can be used when writing to the reference, and another for the most specific type that can be read from it. This can be easily expressed using our readable and writable reference types:

$$\mathrm{ref}\ \alpha, \beta \triangleq \mathrm{ref}_{-}\alpha \cup \mathrm{ref}_{+}\beta$$

Applying this in the context of object calculi would lead not only to more fine-grained subtyping but also to simplifications [21, Appendix A]. In particular, the variance annotations would no longer be needed, and the complex and seemingly ad-hoc rules for subtyping object types [2] would be replaced by the simpler:

$$(\textsc{SubObj})\ \frac{E \subseteq D \quad \forall e \in E.\ B_e {\leqslant} A_e \ \wedge\ A'_e {\leqslant} B'_e}{\left[\mathrm{m}_d : A_d, A'_d\right]_{d \in D} \leqslant \left[\mathrm{m}_e : B_e, B'_e\right]_{e \in E}}$$

We plan to further investigate this. For now, it is worth noting that constructing semantic models has provided us with deeper insights about the underlying calculus than purely syntactical arguments.

Accommodating self types. Since our type system features recursive and bounded existential types, self types (*i.e.* recursive object types with 'proper' subtyping) can be accommodated via an encoding. While we have not checked the details, we expect that a treatment of self types can be achieved even more directly in the model: given monotonic and non-expansive type constructors F_d, we would define a self type $[\mathrm{m}_d :_{\nu_d} F_d]_{d \in D}$ as in Definition 3.19 (Object types) but changing condition (2) to

$$\forall d \in D.\ \langle k, \Psi, l_d \rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to F_d(\alpha')).$$

However, unlike in the functional case, for the imperative calculus a remaining problem is that updated methods cannot take advantage of the self types. Abadi and Cardelli give a modified imperative object calculus with a more complex typing rule for method update that fixes the problem [2, Chapter 17]. Incorporating these more general ideas is not straightforward since the underlying calculus and its operational semantics would change. While we believe that the step-indexing technique is flexible enough to handle this, we leave this for future work.

Very modal models. Appel *et al.* recently proposed a new semantic model which improves the one by Ahmed *et al.* by considering more fine-grained semantic types [11]. It would be interesting to see whether we can more naturally accommodate imperative objects in this more general model.

More realistic languages. Even though the imperative object calculus provides a good framework for theoretical experiments, it is different from any of the object-oriented languages used in practice. It would probably be more complicated, but also more useful, to construct a step-indexed model for a real class-based object-oriented programming language, or at least for a less abstract simplification thereof (*e.g.* MJ [15]).

More than types. The step-indexing technique has already been employed for more general reasoning about programs, not only for type safety proofs. Based on previous work by Appel and McAllester [10], Ahmed built a step-indexed partial equivalence relation model for the lambda calculus with recursive and impredicative quantified types. She showed that her relational interpretation of types captures exactly contextual equivalence [6].

Benton also used step-indexing as a technical device, together with a notion of orthogonality relating expressions to contexts, to show the soundness of a compositional program logic for a very simple stack-based abstract machine [13]. He also employed step-indexing in a Floyd-Hoare-style framework based on relational parametricity for the specification and verification of machine code programs [14].

We hope that our work paves the way for more compelling, semantic investigations of program logics for the imperative object calculus. We think that it might be possible to use a step-indexed model to prove the soundness of more expressive program logics for this calculus. Unfortunately, in order to achieve this goal, we are still lacking a good understanding of how dependent types can be modeled in the step-indexed framework, in the presence of side-effects like non-termination and state.

References

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 332–341. IEEE Computer Society Press, 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Conference record of the 23rd Symposium on Principles of Programming Languages*, pages 396–409. ACM Press, 1996.

[4] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, Lecture Notes in Computer Science, pages 11–41. Springer, 2004.

[5] Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[6] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[7] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of 17th Annual IEEE Symposium Logic in Computer Science*, pages 75–86. IEEE Computer Society Press, 2002.

[8] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Princeton University, January 2003.

[9] Andrew W. Appel and Amy P. Felty. A Semantic Model of Types and Machine Instructions for Proof-Carrying Code. In *POPL 2000: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, Boston, MA, Jan 2000. ACM Press.

[10] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[11] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007.

[12] Andrew W. Appel, Christopher Richards, and Kedar Swadi. A kind system for typed machine language. Technical report, Princeton University, September 2002.

[13] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In Zoltan Esik, editor, *Asian Symposium on Programming Languages and Systems APLAS'05*, volume 3780 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2005.

[14] Nick Benton. Abstracting allocation: the new new thing. In *Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

[15] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[16] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[17] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. *Theoretical Computer Science*, 2007. Special issue in honor of Mario Coppo, Mariangiola Dezani-Ciancaglini and Simona Ronchi della Rocca. To appear.

[18] Cormac Flanagan, Stephen Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *Proc. FOOL/WOOD'06*, 2006.

[19] Andrew D. Gordon, Paul D. Hankin, and Søren. B. Lassen.

Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science (FST+TCS'97)*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer, 1997.

[20] Cătălin Hriţcu. A step-indexed semantic model of types for the functional object calculus. Master's thesis, Advisor: Jan Schwinghammer, Programming Systems Lab, Saarland University, May 2007.

[21] Cătălin Hriţcu and Jan Schwinghammer. A step-indexed semantics of imperative objects. Extended version, Programming Systems Lab, Saarland University, December 2007. Available at: `http://www.ps.uni-sb.de/Papers`.

[22] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theoretical Computer Science*, 338(1-3):17–63, 2005.

[23] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[24] Paul Blain Levy. Possible world semantics for general storage in call-by-value. In Julian Bradfield, editor, *Proceedings of the 16th Workshop on Computer Science Logic (CSL'02)*, volume 2471 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2002.

[25] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):99–124, 1991.

[26] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[27] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[28] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[29] Bernhard Reus. Modular semantics and logics of classes. In Matthias Baatz and Johann A. Makowsky, editors, *17th Workshop on Computer Science Logic (CSL'03)*, volume 2803 of *Lecture Notes in Computer Science*, pages 456–469. Springer, 2003.

[30] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[31] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316:191–213, 2004.

[32] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173-233, Birkhäuser, 1997.

[33] Jan Schwinghammer. *Reasoning about Denotations of Recursive Objects*. PhD thesis, Department of Informatics, University of Sussex, 2006.

[34] Kedar N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, July 2003.

[35] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.