Reasoning about Denotations of Recursive Objects

Jan Schwinghammer

Submitted for the degree of D. Phil.

University of Sussex

September, 2005

Declaration

I hereby declare that this thesis has not been previously submitted, either in the same or different form, to this or any other university for a degree.

Signature:

Preface

The contents of Part II were obtained as the result of joint research with my supervisor, Dr. Bernhard Reus. A preliminary version of this chapter appeared as a University of Sussex technical report (Reus and Schwinghammer 2004), and an extended abstract of our results has been published in the conference proceedings of the European Symposium on Programming (Reus and Schwinghammer 2005). The full version appears as (Reus and Schwinghammer 2006a).

Reasoning about Denotations of Recursive Objects

Jan Schwinghammer

Summary

This thesis is concerned with reasoning about stateful programs where storage of values of all types is possible, including those of higher type. This feature is often referred to as higher-order store; examples are the objects in the object calculus of Abadi and Cardelli as well as the general references of languages such as Standard ML and Scheme. Higher-order store introduces recursion "through the store" to the language, and requires the semantic domain to be defined by a mixed-variant recursive equation. Using domain-theoretic techniques we investigate semantics and logics of languages with higher-order store, with particular emphasis on object-oriented languages where subtyping introduces additional complexity.

The thesis is divided into three parts. The first one surveys some key technical results from domain theory, and summarises various proposals of semantic interpretations of both functional and imperative objects found in the literature. The object calculus that is considered throughout Parts II and III is presented, including its operational and denotational semantics.

Part II presents a denotational semantics for Abadi and Leino's logic of objects. Our soundness proof provides an insightful alternative to the original proof of Abadi and Leino which was given with respect to an operational semantics. By separating validity from derivability in the proof system, we clarify the meaning of specifications of the logic. The logic is also extended by a notion of recursive specification and appropriate proof rules are introduced.

In the final part, the problem of finding a typed model of the object calculus is addressed. Starting from a model of an ML-like language recently presented by Levy, we add subtyping to obtain a semantic model that is sufficiently rich to interpret imperative objects. The semantics is presented as a possible worlds model that explicates the allocation of new memory; subtyping is interpreted using coercion maps. After establishing coherence by extending a method due to Reynolds, a number of non-trivial programs are shown to meet their specifications.

Submitted for the degree of D. Phil.

University of Sussex

September, 2005

Abstract

Most programming languages provide constructs for accessing and manipulating state. In fact, languages such as Standard ML even allow the storage of executable code – in the form of function closures – by implementing "general references". This feature allows us to write general recursive functions by making the recursive calls in a program "through the store", a technique that is known as *recursion through the store* or *back-patching* (Landin 1964). Similarly, object-based languages provide for objects being created on-the-fly, and arbitrary method code needs to be kept in the heap store.

This thesis is concerned with reasoning about stateful programs that use higher-order storage. In modelling such languages, higher-order store requires the semantic domain to be defined by a mixed-variant recursive equation. Our underlying assumption here is that *domain theory* provides an adequate means to deal with the inherent recursion introduced by higher-order store. Using domain-theoretic techniques we investigate semantics and logics of languages with higher-order store, with particular emphasis on object-oriented languages where subtyping introduces additional complexity.

The thesis is divided into three parts. The first one begins by reviewing the key technical results from domain theory that are used in subsequent chapters: A number of constructions on complete partial orders, solution of recursive domain equations in bilimit-compact categories (Smyth and Plotkin 1982; Levy 2004), and relational structures and invariant relations (Pitts 1996). Then, after summarising various proposals from the literature for the semantic interpretation of (both functional and imperative) objects, a variant of the object calculus as an elementary formalisation of object-based languages is presented (Abadi and Cardelli 1996). This is the model of objects used in Parts II and III of the thesis, and we recall its type system as well as its operational and denotational semantics. Since the store model differs slightly from the one used in the original formulation of the calculus, we show agreement between operational and denotational semantics by establishing adequacy. While the result itself is not surprising, and its proof proceeds in a standard way, this provides a first illustration of the use of relational structures.

In the second part of the thesis, a denotational semantics for Abadi and Leino's (1997, 2004) logic of objects is given. The logic has a compositional proof system that refines the type system. Our semantics is based on the untyped cpo model of objects considered in previous work of Reus and Streicher (2002, 2004). We extend their work in order to deal with mutually recursive objects, the dynamic allocation of new objects on the heap, and a notion of subspecification featured by the logic.

Our soundness proof for the logic provides an insightful alternative to the original proof by Abadi and Leino which was given with respect to an operational semantics. By clearly separating validity from derivability in the proof system, we clarify the meaning

of specifications of the logic: Every specification denotes a predicate over the denotations of programs. Moreover, the denotational analysis of the logic justifies some of the restrictions made in the logic: Semantically, they guarantee the well-definedness of specifications that talk about higher-order store. Further, the denotational setting is used to extend the logic with recursive specifications which provide a way to reason about operations on recursively defined data structures. After introducing corresponding proof rules we demonstrate that our soundness proof can be adapted to cover this extension.

In Part III, the problem of finding a typed model of the object calculus is addressed. Our apporach is indirect: Starting from a typed model for an SML-like language recently presented by Levy (2002, 2004), we add subtyping to obtain a model of call-by-value languages with higher-order functions, records, references to values of arbitrary type and structural subtyping. Adapting the semantics from (Kamin and Reddy 1994) this provides for an interpretation of the object calculus, too.

The semantics is presented as a possible worlds model that explicates the allocation of new memory; subtyping is interpreted using coercion maps. Coherence of this semantics is established by a method due to Reynolds (2002b), where a logical relation between the typed and an untyped model is used. The coherence theorem is the main technical result in this part. As a corollary to the coherence theorem, the logical relation gives rise to a per model of higher-order storage. To the best of our knowledge this is the first such model; we discuss how it differs from previous per models in order to achieve type safety in the presence of destructive updates.

Finally, we illustrate the potential gains of reasoning in an (intrinsically) typed model. Unfortunately our attempts at providing a logic for this language (similar to the one of Abadi and Leino for the object calculus) did not succeed. Nevertheless the model may be used to prove correctness of several non-trivial example programs in an ad-hoc fashion, by reasoning directly about their denotations. The examples we give include: Recursive functions, methods that use recursion through *self*, a simple call-back mechanism, and functions and objects that use recursion through the store. Compared to similar correctness proofs considered in earlier work in the untyped model (Reus and Streicher 2004), specifications can often be written in a more concise way, taking advantage of the fact that all programs can be assumed to be type correct.

Some preliminary results about local reasoning for higher-order store are given at the end of the thesis, in the form of a frame property known from recent work on separation logic (Reynolds 2002a).

Acknowledgements

I am indebted to my supervisor, Dr. Bernhard Reus, without whom this thesis would presumably not exist. I wish to thank him for proposing the topic of this thesis to me, and for teaching, listening and patiently answering my many questions. He also provided much appreciated encouragement and was persistent at times when our research did not seem to progress.

I want to thank my examiners, Prof. Guy McCusker and Prof. Uday Reddy, for their careful reading and insightful questions that helped to improve correctness and presentation. I am grateful to the members of my Thesis Committee, Prof. Guy McCusker and Prof. Vladimiro Sassone, for their interest and their constructive and relevant feedback on my work. Thanks as well to the other faculty, students and visitors who made the Foundations group (and the Theory Lab in particular) such an enjoyable place to work. I would also like to thank my good friends Dr. Manuel Bodirsky, Tobias Gärtner, and Dr. Timo von Oertzen, who all read and commented on previous versions of this text, and Damiano Macedonio, who explained to me the subtleties of intuitionistic logic and Brunched Implications.

For some very interesting discussions about Abadi and Leino's logic, and about completeness issues in general, I thank Prof. Thomas Streicher. I must also thank Dr. Paul B. Levy for explaining his model construction to me during a visit to Sussex, and for pointing out a flaw in an earlier version of Chapter 8.

Thanks also to Prof. Dr. Gert Smolka and the members of the Programming Systems Lab in Saarbrücken, for their hospitality during several short visits, and for providing me with the opportunity to finish my thesis at the lab. Andreas Rossberg in particular gave some useful hints about ML polymorphism and could confirm that recursion through the store *is* used in real programs.

Iris Kersten and Dr. Gregor Erbach provided friendship, bed, breakfast and generous help in many ways when desperately needed, thank you. Thanks to Dr. Martin Coleman and Dr. Sebastian Rasinger for being good friends and housemates for many years, and to all my running and cycling friends for taking my mind off work every now and then.

I gratefully acknowledge the support for this research provided by the Engineering and Physical Sciences Research Council of the United Kingdom, through grant GR/R65190/01, "Programming Logics for Denotations of Recursive Objects".

Finally, many thanks to my family for their support throughout.

Jan Schwinghammer,
Saarbrücken, October 13, 2006.

Contents

List of Tables

Chapter 1

Introduction and Background

The theory of functional programming languages and lambda calculus is well-developed. In computing practice, however, almost all widely used languages contain constructs for *imperative programming*, such as mutable variables, pointers and references.

The mathematical modelling of first-order WHILE languages with integer storage poses no particular problems; Floyd-Hoare logic and similar calculi provide formal frameworks for specification and correctness proofs of programs (Cousot 1990). But already the combination of block-allocated integer store and higher-order functions, as modelled by the IDEALIZED ALGOL family of languages (Reynolds 1981), is considerably more complex. Many programming languages go much further and allow the storage of arbitrary, *executable code*. The heap store of such languages is often referred to as *higher-order store*.

It has been known for a long time (Landin 1964) that higher-order store can be used to implement recursion, by a technique called *recursion through the store*, or *back-patching*. Consider the program fragment:

$$\begin{array}{l} \mathsf{let}\ r : \mathsf{ref}\ (\mathsf{int} \Rightarrow \mathsf{int}) = \mathsf{new}\ \lambda n.\, n\ \mathsf{in} \\ \mathsf{let}\ f : \mathsf{int} \Rightarrow \mathsf{int} = \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (\mathsf{deref}\ r)(n-1) \\ \mathsf{in}\ r := f;\ \ldots \end{array} \tag{1.1}$$

A reference r is created and initially contains the identity function on the integers int. The function f declared next simply calls the function that r refers to in the else-branch. This means that, after the final update, f calls itself recursively and one (rightly) expects that f computes the factorial. An *informal* argument of this fact may be along the following lines: *If the reference r contains a function that computes the factorial, then f computes the factorial. Hence, after the assignment of f itself to r in line 3, f computes the factorial.*

Because of the inherent circularity of this description, the validity of this argument is not at all obvious. In fact, consider the similar program where the definition of f is replaced by

$$\begin{array}{l} \mathsf{let}\ f : \mathsf{int} \Rightarrow \mathsf{int} = \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ (\mathsf{deref}\ r)(n-1) \\ \mathsf{in}\ r := f;\ \ldots \end{array} \tag{1.2}$$

That is, f is like the function stored in the reference r but shifted by 1. Call a function g on the natural numbers *globally non-constant 0* (gnc) if it is not eventually constant 0, i.e., if for all n there is $m \geq n$ such that $g(m) \neq 0$. One might be tempted to reason as before and argue that r contains a gnc function after the assignment: *If the reference r contains a gnc function, then f is gnc. Hence, after the assignment of f itself to r, f is gnc.* However, operational intuition tells us that f is indeed 0 everywhere. We must conclude that the soundness of this reasoning principle is guaranteed only under additional assumptions.

Abadi and Cardelli's (1996) calculus of imperative objects also allows one to write programs similar to (1.1):

$$\left[\ \mathsf{fac} = \varsigma(y)\lambda n.\,\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times y.\mathsf{fac}(n-1)\ \right] \tag{1.3}$$

The program defines an object with a single method fac. The *self parameter* y is bound to the object itself, providing a way to call the method recursively in the method body. According to the semantics of (Abadi and Cardelli 1996) the recursive binding of the self parameter is through the store. A program logic, due to Abadi and Leino (1997), provides inference rules that allow us to prove correctness of programs using recursion through the store. Again, soundness of the logic is not obvious.

This thesis is concerned with semantics and logics of programming languages with higher-order store. In particular, we investigate higher-order store in the context of languages with subtyping, and object-oriented languages. *Denotational semantics* will facilitate formal proofs of programs such as (1.1) and (1.3), and provide an explanation why the reasoning in (1.2) failed. Our contributions are in two areas:

A Logic for an Object Calculus. A program logic for the object calculus, first presented in (Abadi and Leino 1997), allows for reasoning about programs using recursion through the store. Using an untyped domain-theoretic model of the calculus, we use techniques similar to those of (Reus and Streicher 2004) to prove *soundness* of this logic. Starting from our semantics, the logic is also extended with *recursive specifications* that refine recursive types.

A Model of Higher-order Store and Subtyping. We address the problem of constructing a *typed model* of the object calculus. A typed model of a lambda calculus with general references, due to Levy (2002), is extended by the introduction of subtyping. It is shown how to soundly model this semantically, by proving *coherence*. Using a well-known encoding of imperative objects (Reddy 1988; Kamin and Reddy 1994) this gives rise to a typed model of Abadi and Cardelli's object calculus.

The remainder of this chapter contains an informal overview of the material, to set our results in the wider context.

1.1 Objects, Classes and Imperative Object Calculus

Objects consist of *fields* providing local state, and *methods* that may modify this state. In *class-based languages*, *classes* provide templates from which objects are instantiated; a class describes the *interface* and *behaviour* of its instance objects. Classes separate

the method code from objects, the class *name* is used to select the appropriate code for method invocations from the *class table*.

Object-based languages are based on the notions of *prototype object* and *cloning* (Abadi and Cardelli 1996, Chapter 4). Their distinguishing feature – compared to class-based object-oriented languages – is that objects are assembled on-the-fly. Rather than instantiating from a class of a fixed class table (which means there are only finitely many classes), there are constructs for the creation and customisation of individual objects. This includes the dynamic adaptation of methods and has the consequence that each object contains its own set of methods.

Abadi and Cardelli (1996) present a formal model of object-based programming. A family of languages with various type systems and semantics is introduced, which are often simply referred to as *object calculus*. Intuitively, the expression

$$[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J}$$

allocates a new object on the heap, with fields f_i and methods m_j. The expression $a.\mathsf{f}_i$ returns the value of field f_i, the assignment $a.\mathsf{f}_i := b$ updates its value. Method invocation, $a.\mathsf{m}_j()$, executes the method body b_j; in the body, the variable y_j refers to the host object a (this variable is typically called `self` or `this` in object-oriented languages). Upon invocation, the self parameter y_j of the method m_j is therefore bound to the location of the object in the heap. The expression `clone` a produces a (shallow) copy of the object a in the store. Certain aspects of class-based languages can also be simulated in the object calculus, using *compilation techniques* (Abadi and Cardelli 1996, Chapters 8, 11 and 12).

One of the aims of (Abadi and Cardelli 1996) was the investigation of type systems for object-oriented languages, in particular the interaction between *subtyping* and the recursion through `self`. Later work (Abadi and Leino 1997) refined a simple type system to obtain a *program logic* for the imperative object calculus.

We shall focus on reasoning about object calculus programs in this thesis. Abadi and Leino's logic in particular is our topic of study in Part II.

1.2 Higher-order Store

Higher-order store, by which we mean a store[1] that may contain higher-order values, comes in various guises. Many widely used programming languages, as well as foundational calculi, contain constructs that give rise to store that is higher-order.

General References

Higher-order store appears in the form of *general references* in typed higher-order languages like Standard ML (Milner et al. 1997). References in ML denote (typed) locations, or *cells*, in the heap. References can be created to values of arbitrary types: If A is any type, including functional types $B \Rightarrow B'$, then `ref` A denotes the type of references containing values of type A. Operationally, this amounts to storage cells containing *function closures*.

The program (1.1) illustrates general references. Indeed, apart from minor notational differences (1.1) is valid Standard ML code. Moreover, similar examples show that general

[1] We use the terms *store* and *heap* synonymously

references allow us to write general recursive functions, even in the absence of a fixed point combinator and despite simple typing. This contrasts with simply typed lambda calculus that is well-known to be strongly normalising. The semantics of recursively defined functions in the language Scheme (Abelson et al. 1998; Honsell, Pravato, and Rocca 1998) is actually stated in terms of *recursion through the store*, following the pattern of (1.1).

A surprising observation is that the combination of references for *basic types* such as int and *storable commands* (in ML, these correspond to functions $\mathbf{1} \Rightarrow \mathbf{1}$ on the one-element type, $\mathbf{1}$) are already sufficient to simulate general references (Guy McCusker, personal communication). The idea is that a function of type $A \Rightarrow B$ can be simulated by a command, corresponding to the function body, that accesses the argument via a reference of type ref A, and returns the result via a reference of type ref B.

Function Pointers

In low-level languages such as C (Kernighan and Ritchie 1988) and C++ (Stroustrup 2000), the concept of *function pointers* amounts to a form of higher-order store. For instance, a C function pointer *ptr2fn* is defined by

$$\mathsf{int}\ (*\mathit{ptr2fn})(\mathsf{int}) = \&f;$$

where f is a function that takes an argument of type int and returns an int, and '&' is the *address operator*. The pointer *ptr2fn* is treated like any other C pointer: It can be assigned to, can be dereferenced, and can be passed as argument and return value in function calls. An analogue of (1.1) can be written in C as follows.

$$\begin{array}{l}\mathsf{int}\ (*\mathit{ptr2fn})(\mathsf{int}\ n) = \mathsf{NULL};\\ \mathsf{int}\ \mathit{fac}(\mathsf{int}\ n)\ \{\ \mathsf{if}\ (n == 0)\ \mathsf{return}\ 1;\ \mathsf{else}\ \mathsf{return}\ (n \times ((*\mathit{ptr2fn})(n-1)));\ \}\\ \mathsf{int}\ \mathit{main}(\ldots)\ \{\ \mathit{ptr2fn} = \&\mathit{fac};\ \ldots\}\end{array}$$

Object-Based Languages

The ability to dynamically create objects in object-based languages has the consequence that each object contains its own set of methods. Since objects are stored in the heap, this leads to a higher-order store.

Here is yet another variant of the factorial, written in Abadi and Cardelli's (1996) imperative object calculus.

$$\begin{array}{l}\mathsf{let}\ a : A = [\mathsf{fac} = \varsigma(y)\lambda n.\, n]\ \mathsf{in}\\ \mathsf{let}\ b : B = [\mathsf{f} = a,\ \mathsf{fac} = \varsigma(y)\lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times y.\mathsf{f.fac}(n-1)]\\ \mathsf{in}\ b.\mathsf{f} := b;\ \ldots\end{array} \qquad (1.4)$$

where $A \equiv [\mathsf{fac}{:}\mathsf{int}\Rightarrow\mathsf{int}]$ and $B \equiv [\mathsf{f}{:}A, \mathsf{fac}{:}\mathsf{int}\Rightarrow\mathsf{int}]$. The difference to (1.3) is that the recursion in b's method fac is through the field f rather than directly through the self parameter y. The assignment in the third line is acceptable because the types A and B are in *subtype relation*, so that every object of type B may be used where objects of type A are expected (by "forgetting" f, see Section 1.4 below).

1.3 Denotational Semantics

Reasoning about the behaviour of programs, purely in terms of their syntax and operational semantics, is often difficult. An alternative approach to proving program properties was initiated by the work of Scott and Strachey on denotational models of programming languages (Strachey 1966; Scott 1993; Scott and Strachey 1971; Scott 1972). To quote Fiore et al. (1996), a denotational semantics intends "to bring out subtle issues in language design, to derive new reasoning principles, and to develop an intuitive abstract model of the programming language under consideration so as to aid program development."

Each term a of the language is assigned a mathematical object $[\![a]\!]$, its *denotation* or *meaning*. The mapping from programs to their denotations is usually compositional in the sense that the denotation of an expression is determined by the denotations of its subexpressions. Instead of reasoning about concrete program executions, one can then use (mathematical) properties to reason about the program's denotation.

There must be a good "fit" between denotational model and concrete executions, usually formalised in terms of operationally defined *observations* and *observational equivalence*, and the statement of *adequacy* with respect to these observations (Winskel 1993). Adequacy asserts that reasoning about the denotations can soundly be transferred back to the operational behaviour of programs (although it may not be *complete*). However, in this thesis we shall focus on denotational semantics. Therefore we will not be overly concerned with the relation between operational and denotational semantics.

1.3.1 Domain-theoretic Models

The possibility of non-termination of programs induces a natural order on (partial) functions: f is less or equal g if and only if g converges on more inputs, and moreover f and g agree on all inputs where f converges. Ordered structures therefore arise naturally in the semantics of programming languages. The intuition is then that the ordering describes the *information content* of an element (Plotkin 1983).

Domain-theoretic models are partially ordered sets that are widely used in denotational semantics. They are required to be *complete* in that least upper bounds exist for sets of elements that arise by approximating recursive functions, looping constructs and similar computational phenomena. Maps between domains are required to be monotonic and continuous (preserve the least upper bounds). Plotkin's (1983) lecture notes contain intuitive motivation for these definitions. Technically speaking, continuity entails that (least) fixed points of endomaps exist.

Many classes of domains also permit solutions to domain equations $X \cong F(X)$ which allows one to define domains recursively (Smyth and Plotkin 1982). A well-known example is the solution to $X \cong X \multimap X$ that has been used to give a semantics to untyped lambda calculus. For our purpose of investigating higher-order storage the ability to solve recursive domain equations is an important property, for the following reason. Let St be a domain of stores and let Z be the set of integers. Imperative functions may have side-

effects on the store, so semantically a function on the integers is a partial map

$$f : Z \times St \rightharpoonup Z \times St$$

Since such functions may also be stored in a higher-order store $s \in St$, say at location l, we must have $s.l \in (Z \times St \rightharpoonup Z \times St)$. In other words, the definition of St necessitates the solution of a recursive equation.

As a consequence, program properties are generally expressed by (recursively defined) predicates on such recursively defined domains. (Pitts 1996) introduces a framework of relations over domains, a slight generalisation of which is presented in Chapter 2 following the ideas of Levy (2004). This provides the technical machinery for reasoning about programs with higher-order storage.

We follow the terminology of Reynolds (2002b) and call a denotational model of a typed language an *intrinsic* semantics if meaning is given to derivations of typing judgements $\Gamma \triangleright a : A$, rather than to terms a, with the consequence that

- ill-typed phrases are meaningless,
- terms satisfying several judgements will be assigned several meanings, and
- coherence between the meaning of several derivations of the *same* judgement must be established.

In contrast to intrinsic semantics, an *extrinsic* semantics gives meaning to *all* terms, i.e., one starts with a model of an untyped language. Types and typing judgements are interpreted as, e.g., predicates or partial equivalence relations over the model (cf. Section 1.4).

1.3.2 Denotational Semantics of Storage

It is not hard to obtain denotational models of first-order imperative WHILE languages, where the store is usually modelled by (finite) maps from locations to values. But the semantics of storage becomes quite involved once we look beyond such simple WHILE languages. For example the combination of (call-by-name parameter-passing) higher-order procedures and *block-allocated storage*, exemplified by ALGOL-like languages, poses several challenging problems for obtaining "good" models (Reynolds 1981): Naive models of storage allocation fail to satisfy such basic equivalences as the *garbage collection equation*

$$[\![\mathsf{let}\ x = \mathsf{new}\ e\ \mathsf{in}\ c]\!] = [\![c]\!]$$

when x is not free in c (Meyer and Sieber 1988).

In many proposals of semantic models that more adequately treat local variables and allocation of memory, the denotation of types is parameterized by *store shapes*, yielding a *possible-worlds* semantics of types. *Functor categories* provide the necessary machinery to construct such models (Oles 1982; O'Hearn and Tennent 1997). By imposing a strong notion of uniformity, *parametricity*, one can explain the constrained action of non-local procedures on local state (O'Hearn and Tennent 1995; Sieber 1994). This provides a

formal link to parametric polymorphism and its application to abstract data types and independence of data representation (O'Hearn and Reynolds 2000; Reynolds 1983; Mitchell and Plotkin 1988; Mitchell 1991).

In the *co-algebraic modelling of objects* (Jacobs 1996; Jacobs and Rutten 1997; Jacobs and Poll 2003), classes are viewed as coalgebras $S \longrightarrow T(S)$ consisting of a state space S and an implementation of the methods T (as *transition function*). In this reading, an object is modelled as an element of S; conceptually, the state can only be observed by interaction via T. Bisimulation can be used to show indistinguishability by T-observations between objects, possibly even belonging to different classes. However, modelling object-based languages where there is no proper distinction between class and object appears to be beyond current co-algebraic techniques. Reddy's event-based semantics of objects (where references to objects are not allowed) has a similar flavour (Reddy 2002; Reddy 1996).

Other variations of storable values have been considered; Tennent and Ghica (2000) survey the history and development of semantic models of storage. Pitts and Stark investigate the combination of call-by-value functions and dynamically created *names* (Pitts and Stark 1993; Pitts 1996) and dynamically created integer references (Pitts and Stark 1998). Functor categories prove a useful tool in this instance, too. Building on these results, recent work of Reddy and Yang (2004), and Benton and Leperchey (2005) has provided a fairly successful semantic account of local state in languages with dynamically allocated heap memory including pointers. Reddy and Yang use a parametric model to express the inaccessibility of non-local, private store, employing ideas from separation logic (O'Hearn, Reynolds, and Yang 2001) to track the visibility and leaking of pointers. Benton and Leperchey develop similar ideas on top of a model built in the category of FM-cpos (Shinwell and Pitts 2005). FM-sets and -cpos provide an elegant way of expressing *freshness*, which is used to neatly model the irrelevance of actual location names.

Higher-order store is less well-understood. An investigation of locality, data abstraction and leakage of references in ML-like languages without restrictions on the storable types is still largely missing. Paul Levy recently presented a possible worlds model that provides the starting point for the typed semantics of objects we provide in the final part of this thesis (Levy 2004; Levy 2002). The model reflects the dynamic allocation aspects but its use of a *global store* prevents reasoning about private variables and data abstraction.

Besides Levy's work we are aware of only few other (typed) semantic models of higher-order store in the literature. The models (Abramsky, Honda, and McCusker 1998; Laird 2003) use games semantics and are *storeless* and not location-based, i.e., the store is modelled only indirectly via possible program behaviours. They do not immediately give rise to strong reasoning principles. In particular, no analogue of the minimal invariant property has been developed, on which Pitts' (1996) theory of relational properties is based. We found these necessary, for instance, to establish the existence of the logical relation in Section 8.6.

Ahmed, Appel, and Virga (2002) construct a model with a more operational flavour: The semantics of types is obtained by approximating absence of type errors in a reduction

semantics; soundness of this construction follows from an encoding into type theory. Again we do not see how strong reasoning principles can be obtained. Jeffrey and Rathke (2002) provide a model of the object calculus in terms of interaction traces, very much in the spirit of games semantics. Apart from Jeffrey and Rathke's semantics, none of these models treats subtyping.

1.4 Subtyping

Subtyping is a useful and practical mechanism to increase the expressiveness of typed languages: It allows terms of type A to be placed in contexts that expect values of type B, whenever A is a *subtype* of B (written $A \preceq B$). A common subtype relation is between the types int of integers and real of floating point numbers. Postulating $\mathsf{int} \preceq \mathsf{real}$ means that expressions of type int may be used in place of expressions of type real. In the type system, this substitutability is realised by the inference scheme

$$\frac{\Gamma \rhd a : A \quad A \preceq B}{\Gamma \rhd a : B}$$

called the *subsumption rule*.

Given a set of subtypings $A \preceq B$ on base types, the subtype relation can be lifted to higher types in a standard way, following the type structure. For instance, a function of type $A \Rightarrow B$ expects an input of type A, and returns a result of type B. Therefore it may safely be applied to elements of a subtype of A, and used in contexts requiring at most something of type B. Thus, we expect a function type $A \Rightarrow B$ to be a subtype of another such type $A' \Rightarrow B'$ if $A' \preceq A$ and $B \preceq B'$. Similarly, the only operation on (functional) records is the selection of fields. Therefore a record of type $\{\mathsf{f}_i : A_i\}_{i \in I}$ can safely be placed in contexts where only a subset of the fields is accessed. Hence, a record type $\{\mathsf{f}_i : A_i\}_{i \in I}$ should be a subtype of a record type $\{\mathsf{f}_j : A_j\}_{j \in J}$ provided $J \subseteq I$. The subtype relation that we use for the language considered in Part III is induced by these two principles.

Subtyping where the subtype relation is extended from base to higher types in this way is called *structural subtyping*. It is the natural notion of (interface) subtyping in object-based languages. In contrast, many class-based object-oriented languages implement *nominal* subtyping where the subtype relation is derived solely from a relation on type names, according to the subclass hierarchy in the class table. We will be mainly concerned with object-based languages and therefore with structural subtyping.

With regards to the denotational semantics we wish to exhibit models of structural subtyping that justify the informal description just given. More precisely, this amounts to establishing soundness of the subsumption rule with respect to an interpretation of types in a model. Models of subtyping can be divided into two classes: Ones where the subtype relation is interpreted as a suitable form of inclusion, and ones where $A \preceq B$ corresponds to a *coercion map* c_{AB} from the interpretation of A to the interpretation of B.

1.4.1 Containment Interpretation of Subtyping

The most intuitive view of subtyping is that subtyping means inclusion: If $[\![A]\!]$ is the set of values of type A, and $[\![B]\!]$ is the set of values of type B, then $[\![A]\!] \subseteq [\![B]\!]$ holds whenever $A \preceq B$. Soundness of subtyping on functions and record types, as described above, follows immediately from such an interpretation of subtyping.

A semantics for typed languages can be obtained by starting with a model for the untyped language. In a second step, types are interpreted by choosing suitable subsets of the untyped model. As a well-known example, for lambda calculus a denotational semantics can be given in a *universal domain* U where there are (partial continuous) maps e and p,

$$[U \rightharpoonup U] \underset{p}{\overset{e}{\rightleftarrows}} U$$

between U and the space of partial continuous functions on U, satisfying $p \circ e = \mathsf{id}_{[U \rightharpoonup U]}$. Types are interpreted as subsets of U, defined inductively by

$$[\![A \Rightarrow B]\!] \stackrel{def}{=} \{f \in U \mid \forall u \in U.\, u \in [\![A]\!] \wedge p(f)(u)\downarrow \implies p(f)(u) \in [\![B]\!]\} \tag{1.5}$$

where $p(f)(u)\downarrow$ means that application of the (partial) function $p(f)$ is defined. Indeed this gives rise to a containment interpretation of subtyping. For richer languages the denotations of types may need to be more constrained. For instance, to accommodate fixed point operators, the sets $[\![A]\!]$ typically have to be closed under taking least upper bounds. Reynolds' (2002b) article contains a detailed account of such an interpretation. MacQueen, Plotkin, and Sethi (1986) use a similar setting to model recursive and polymorphic types.

The approach can also be extended from types to more precise specifications, viewed as predicates on the domain of programs. This has been used in LCF, a logic for functional programs (Paulson 1987). Our denotational semantics for Abadi and Leino's logic in Chapter 4 also follows the same idea.

1.4.2 Partial Equivalence Relations

The intuitively pleasing view of subtyping described above has a short-coming with respect to *equational reasoning*. For example, in a functional language with subtyping, the records $\{\mathsf{f} = 1, \mathsf{g} = 1\}$ and $\{\mathsf{f} = 1, \mathsf{g} = 2\}$ are distinct when viewed at type $\{\mathsf{f} : \mathsf{int}, \mathsf{g} : \mathsf{int}\}$. However, they cannot be distinguished at the supertype $\{\mathsf{f} : \mathsf{int}\}$ and should therefore be considered equal. Consequently, equivalence of expressions depends on the type information that a surrounding context possesses. This is not reflected by the containment interpretation where equality is inherited from the underlying domain U and necessarily "untyped".

However, the interpretation can be refined to take typed equational reasoning into account. Models based on partial equivalence relations (pers) reflect this by providing a semantics for both type membership and typed equality (Longo and Moggi 1991; Longley 1995; Mitchell 1996): The denotation of a type A is a partial equivalence relation R_A on U, i.e., an equivalence relation on some subset $U' \subseteq U$ of U. The intuition behind this is

that the domain U' of R_A determines the set of elements of type A, and the equivalence determines when two elements of A are to be considered equal.

For instance, the following per $[\![A \Rightarrow B]\!]$ of functions from $[\![A]\!]$ to $[\![B]\!]$ takes the place of (1.5) in the inductive definition of types:

$$[\![A \Rightarrow B]\!] \stackrel{def}{=} \{\langle f, g\rangle \mid \forall \langle u, v\rangle \in [\![A]\!] . p(f)(u)\downarrow \ \vee\ p(g)(v)\downarrow \implies \langle p(f)(u), p(g)(v)\rangle \in [\![B]\!]\}$$

where $\langle p(f)(u), p(g)(v)\rangle \in [\![B]\!]$ asserts in particular that both $p(f)(u)$ and $p(g)(v)$ are defined. The definition of $[\![A \Rightarrow B]\!]$ guarantees that every f in the domain of this per respects the equivalence classes of $[\![A]\!]$. Inclusion of pers provides for a sound interpretation of subtyping and the subsumption rule.

In fact, pers have proven useful in obtaining models of type theories more expressive than simply typed lambda calculus, most notably including impredicative and parametric polymorphism and type recursion (Amadio 1991; Cardone 1989; Abadi and Plotkin 1990; Freyd, Rosolini, Mulry, and Scott 1992; Phoa 1992; Mitchell and Viswanathan 1996). For many purposes the universal domain U may be replaced by an arbitrary partial combinatory algebra: Several of the cited articles use the natural numbers, viewed as codes for the partial recursive functions.

We will obtain a per model of higher-order store as a corollary to the technical development in Chapter 8.

1.4.3 Coercions

A more general view of subtyping is that a subtype relation $A \preceq B$ determines a *coercion map* $c_{AB} : [\![A]\!] \to [\![B]\!]$ that converts values of type A to values of type B (Mitchell 1996; Mitchell 1984; Reynolds 1980). For example, the internal representations of integers and floating point numbers could be quite different in an implementation; semantically $[\![\mathsf{int}]\!]$ and $[\![\mathsf{real}]\!]$ may well be disjoint. The map $c_{\mathsf{int},\mathsf{real}} : [\![\mathsf{int}]\!] \to [\![\mathsf{real}]\!]$ associated with $\mathsf{int} \preceq \mathsf{real}$ would model the actual conversion that happens in the implementation.

The added flexibility, compared to the containment interpretations, comes at a price. As mentioned in Section 1.3, the meaning of a typing judgement $\Gamma \rhd a : A$ in an intrinsic semantics is usually defined inductively on the *derivation* of this judgement. In the presence of subtyping, in general there are many distinct derivations of the same judgement possible. It becomes necessary to prove *coherence*, stating that if $\mathcal{P}_1(\Gamma \rhd a : A)$ and $\mathcal{P}_2(\Gamma \rhd a : A)$ are two derivations with identical conclusion $\Gamma \rhd a : A$, then their semantics agrees:

$$[\![\mathcal{P}_1(\Gamma \rhd a : A)]\!] = [\![\mathcal{P}_2(\Gamma \rhd a : A)]\!]$$

Typically, coherence is proved by transforming derivations into a normal form while preserving the semantics. Such proofs can be quite involved, even for purely functional languages where non-termination is the only effect, see (Breazu-Tannen, Coquand, Gunter, and Scedrov 1991; Curien and Ghelli 1994; Mitchell 1996) for examples of this method.

An elegant, alternative proof method is presented in (Reynolds 2002b), where a logical relation between an intrinsic coercion model of simply typed lambda calculus with subtyping and a cpo model of the untyped lambda calculus is defined. Reynolds' coherence

proof essentially relies on the fact that the denotations of all derivations of a judgement $\Gamma \rhd a : A$ are related to the untyped denotation $[\![a]\!]$ of a, via the *basic lemma* of logical relations (see (Mitchell 1996), for instance). A family of retractions between the intrinsic semantics and the untyped model is then used to obtain the meaning of $[\![\Gamma \rhd e : A]\!]$ purely in terms of Γ, $[\![a]\!]$ and A, i.e., independent of any particular derivation of the judgement. This method is extended in Chapter 8 to prove coherence for a higher-order language with general references and subtyping.

1.5 Program Logics

Being complementary to the equational reasoning principles mentioned in Section 1.3 above, Floyd-Hoare program logics have proved successful in verifying properties of imperative programs (Floyd 1967; Hoare 1969; Apt 1981; Cousot 1990). Programs are specified with respect to pre- and post-conditions (*before-after* assertions) that describe the effect of a program in terms of changes between the initial and terminal store induced by a run of the program.

A *specification triple* consisting of a program a and pre- and post-condition P and Q, respectively, is traditionally written $\{P\}a\{Q\}$. The assertions P and Q are written in a formal *assertion language* that is often first-order. The treatment of non-termination in the definition of validity of such triples leads to a distinction between *partial* and *total* correctness assertions. We will be interested in partial correctness of programs. For programs a of WHILE languages this means

$$\vDash_{part} \{P\}a\{Q\} \stackrel{def}{\iff} \text{for all stores } \sigma \text{ “satisfying } P\text{”, if } [\![a]\!]\,\sigma\downarrow \text{ then } [\![a]\!]\,\sigma \text{ “satisfies } Q\text{”} \tag{1.6}$$

A formal proof that a program a meets its specification $\{P\} - \{Q\}$ is constructed in a deductive system of axioms and inference rules. For instance, the well-known inference

$$\frac{\{P\}a_1\{R\} \qquad \{R\}a_2\{Q\}}{\{P\}a_1; a_2\{Q\}}$$

allows for reasoning about sequential composition, by finding a suitable logical description R that holds of the intermediate state. This raises questions of *completeness* of such inference systems. A first source of incompleteness could be the inability to express intermediate assertions like R, depending on the assertion language. By including arithmetic, the assertion language becomes *expressive*, i.e., rich enough to express sufficiently many intermediate assertions. But now Gödel's incompleteness theorem implies incompleteness of Hoare logic. Beginning with Cook's (1978) work, proofs of *relative completeness* have been found for various Hoare calculi: Under an expressiveness assumption for the assertion language, completeness of the inference rules is shown relative to a complete system for the assertions.

1.5.1 Procedures, Aliasing and Local Reasoning

Compared to WHILE-languages, the combination of procedures and local variables leads to new problems, most notably that of *aliasing*: Variables x and y are aliased if they

refer to the same location in the store, so that updating one affects the other. An early mechanism for tracking aliased variables are the type systems for interference control (Reynolds 1978; Reynolds 1982). In general, assertions will depend on an environment for free variables, and (1.6) must be rephrased accordingly.

The problem of completeness also becomes more subtle. A result of Clarke (1979) shows that for sufficiently rich programming languages, with local variables and recursive higher-order procedures, no complete proof system exists. However, the situation is different again (i.e., complete systems may exist) if one is willing to sacrifice effective proof checking and moves to a second-order assertion language (Halpern 1984).

More recently, assertion languages with new logical connectives have proven remarkably successful in the context of *local reasoning* in the presence of aliasing and sharing (Reynolds 2002a; O'Hearn, Reynolds, and Yang 2001; Yang 2001). Based on the logic of *bunched implications* (O'Hearn and Pym 1999; Ishtiaq and O'Hearn 2001; Pym, O'Hearn, and Yang 2004), a store satisfies the *separation conjunction* $P * Q$ if it can be split into *disjoint* parts s_1 and s_2 for which P and Q hold respectively. With the help of the separation connective $*$, the *frame rule* that is central to this approach can be stated,

$$\frac{\{P\}a\{Q\}}{\{P * R\}a\{Q * R\}}$$

Thus it allows us to conjoin further invariants R without requiring a new proof about the program a. This is practically important, for example when a is a procedure that is called several times and in different contexts. Local reasoning has been extended to ALGOL-like higher-order languages (with first-order storage) in (Birkedal, Torp-Smith, and Yang 2005). In Chapter 10 we sketch some preliminary ideas about local reasoning and frame rules for a language with dynamically allocated higher-order store.

1.5.2 Logics for Objects, and Abadi and Leino's Logic

A great number of logics for object-oriented languages have been proposed, including (Abadi and Leino 2004; Hensel et al. 1998; Jacobs and Poll 2001; Reddy 2002; Poetzsch-Heffter and Müller 1999; de Boer and Pierik 2004; Pierik and de Boer 2005; de Boer 1999; von Oheimb 2001; Reus et al. 2001). Most of these logics are for class-based languages; Abadi and Leino's (1997, 2004) work is one of the few exceptions.

Since pre- and post-conditions in Hoare logic are separated assertions, *auxiliary variables* are used in order to relate values in initial and terminal states (and results of function calls). Auxiliary variables are variables that appear in assertions but not in programs; they are implicitly universally quantified. For example, in $\{x{=}N\}a\{x{=}N{+}1\}$ the use of (auxiliary variable) N expresses that the effect of executing a is to increment the (program) variable x. Kleymann (1999) contains a thorough discussion of this issue. An alternative approach that is used in Abadi and Leino's logic is the use of a single *transition relation* instead of triples: A transition relation T denotes a predicate over pairs of stores (and perhaps results); possible confusion arising from writing x in an assertion is disambiguated by using *primed* and *unprimed* versions, x' and x, to refer to the value of the variable in terminal and initial state, respectively (Lamport 1994). Thus, Abadi and

Leino's logic for an object calculus term a allows one to derive $\vdash a : T$, meaning that if a terminates, then the transition relation T holds of initial and terminal state and the result.

Transition relations in Abadi-Leino logic cannot talk directly about the methods of objects in the store, i.e., the higher-order part of the store. However, complex results such as an object where each of the methods has to be specified in terms of a transition relation, too, have to be described. For this purpose, *result specifications* of the form $A \equiv \left[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j\right]$ are introduced. As in method bodies, the self parameter y_j allows us to refer to (the specification of) the host object also in specifications. Finally, reasoning is possible under a set of assumptions for non-local variables, so that judgements in fact have the form

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : T :: A$$

The meaning of this judgement is that if a terminates when run in an environment such that each variable x_i satisfies specification A_i, then the transition relation T holds of initial and terminal state and the result, and moreover the result satisfies result specification A.

The following object introduction rule provides the proof principle for objects and lies at the heart of Abadi and Leino's proof system:

$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x_i{:}A_i{::}\ldots \quad \forall i \in I \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j \quad \forall j \in J \end{array}}{\Gamma \vdash \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j\right]_{i\in I, j\in J}{:}A{::}\ldots}$$

In essence, reasoning about the behaviour of a method m_j is done under the assumption that the host object already satisfies its specification A. In this way the recursion through self (implemented by recursion through the store) is treated on the logical level; not surprisingly, proving soundness of this inference rule is nontrivial. The object introduction rule was previously investigated in a denotational framework by (Reus and Streicher 2004). The full inference system, along with a more detailed explanation and a worked example, appear in Chapter 4. Subsequent chapters present its semantics and soundness.

We conclude with a final remark on completeness. Many of the papers for class-based languages cited above prove completeness of the presented proof systems for objects. In view of (Clarke 1979) such results must rely on one of the following properties. Firstly, the assertion language is higher-order logic rather than first-order logic; for example this is the case in (von Oheimb 2001). Secondly, methods are mutually recursive first-order, rather than higher-order, procedures. The latter is possible since objects are instantiated only from a predefined class table, and reasoning under a *closed-world* assumption is achieved.

For the object calculus and Abadi and Leino's logic, neither of these properties hold. Indeed, Abadi-Leino logic is incomplete (Abadi and Leino 2004).

1.6 Further Related Work

In recent years operationally based techniques have become much more sophisticated and provide alternative reasoning principles. Work of Talcott and others (Mason, Smith, and Talcott 1996; Talcott 1998) is concerned with models of (effectful) lambda calculi based on equivalence classes of terms with respect to observational equivalence. Operational analogues of domain-theoretic concepts, such as fixed point induction, are developed to facilitate reasoning about recursive functions. Birkedal and Harper (1999) present a model of recursive types in this way; the construction relies on the fact that the projections in the operational analogue of Pitts' minimal invariant property (see Chapter 2) are syntactically definable. We are not aware of a similar treatment of dynamically allocated higher-order storage, where the definability question for these projections appears to be less obvious.

Bisimulation and trace semantics provide convenient proof principles to establish contextual equivalences for equational reasoning. Gordon, Hankin, and Lassen (1997) and Jeffrey and Rathke (2002) investigate this for the imperative object calculus. As in the games models of storage, the store is described only indirectly in terms of interactions and observations. Therefore these models do not appear directly useful as the basis for program logics in the style of Hoare, where store descriptions form the central concept. However, (Ghica 2001) may provide a good starting point.

In the past year, Honda et al. (Honda, Yoshida, and Berger 2005; Berger, Honda, and Yoshida 2005; Honda 2004) have begun to investigate program logics for higher-order languages, including higher-order storage. Their logics rely on giving names to all intermediate results. This is reminiscent of Abadi and Leino's logic and its use of locations in assertions: Abadi-Leino logic takes advantage of the fact that all complex results are objects, which are kept in the heap store, and are therefore addressable. In contrast to our semantics of Abadi-Leino logic, the interpretation of the language by Honda et al. is in a term model. As a consequence, their soundness proof requires substantial reasoning about contextual equivalence. Another difference is the treatment of recursion through the store: Honda et al. consider total correctness assertions, and the crucial step in a proof of (1.1) can proceed by well-founded induction on a termination order.

1.7 Contributions of this Thesis

The motivating research question that we wish to contribute to is the problem of specification, verification and logic of programs that use higher-order storage.

1.7.1 Our Approach and Results

The underlying assumption in this thesis is that the framework of denotational semantics and domain theory provides adequate techniques for dealing with the recursion inherent in the semantics of higher-order store, both on the level of terms and types.

The treatment of dynamic allocation of memory appears to naturally lead to possible-worlds models of types and specifications. The theoretical framework is thus provided by certain functor categories over complete partially ordered sets.

The first major contribution is the soundness proof for Abadi and Leino's logic in Part II. Previous work (Abadi and Leino 1997) introduced a notion of specification but left open the question of their semantics. Similarly, *store specifications* were introduced as a technical device in their soundness proof, but on a syntactic level only. Based on a simple denotational model of the object calculus, we define a semantics of specifications (Table 4.7) and store specifications (Definition 5.1.7). Well-definedness of the latter requires a non-trivial proof (Lemma 5.1.8) similar to (Reus and Streicher 2004; Pitts 1996); in contrast to loc. cit. existence of a *family* of predicates defined by mutual recursion has to be established. Having a semantics of specifications allows us to define *validity* of specification judgements (Definition 5.2.3) and to prove soundness of the logic (Theorem 5.2.6). Our key lemma (Lemma 5.2.5) strengthens the statement so that an inductive proof goes through.

A second contribution is the extension of this logic with recursive specifications and corresponding proof rules (Tables 6.1 and 6.2). The semantics of recursive specifications is given as a greatest fixed point (Table 6.5 and Lemma 6.2.5). Finite approximations of specifications, known from the work of Amadio and Cardelli (1993) on types, are introduced for recursive specifications to facilitate an inductive proof of the soundness theorem (Lemma 6.3.8 and Theorem 6.3.9).

The next contribution is the construction of an intrinsically typed model for a higher-order language with general references and subtyping. Our technical result here is the coherence theorem (Theorem 8.7.1) which is established by extending Reynolds' (2002b) technique from simply typed lambda calculus to general references: Existence (Theorem 8.6.4) is shown for a logical relation between typed and untyped models, and its fundamental property is proved (Lemma 8.6.7). Bracketing maps are defined (Table 8.10) which establish that the logical relation is left-unique (Theorem 8.6.8). As a corollary, another (extrinsically typed) model of higher-order store is obtained (Section 8.8.1) based on partial equivalence relations.

Using an encoding of objects in this language, we obtain a typed domain-theoretic model of Abadi and Cardelli's imperative object calculus (Table 9.1). To our knowledge no such model has appeared in the literature before. After illustrating reasoning in this model by a number of examples, a variant of the semantic proof rule for objects of (Reus and Streicher 2004) is introduced and proven sound (Rule 9.11, Section 9.3.4).

Our final contribution is the presentation of some preliminary ideas for local reasoning about higher-order store. To this end, a *frame property* (O'Hearn, Reynolds, and Yang 2001) for general references is established (Table 10.1 and Lemma 10.2.6), by trimming down the intrinsic model.

1.7.2 Thesis Outline

Part I of the thesis presents both the mathematical background and the formal programming model used in subsequent chapters. **Chapter 2** reviews the necessary definitions from domain theory that will be used to give denotational semantics. Key results about the solution of recursive domain equations and the well-definedness of recursively de-

fined predicates on such domains are recalled. This chapter could be read on a by-need basis. **Chapter 3** contains an overview of a number of interpretations of objects and object-oriented programming that appeared in the literature, and formally introduces a variant of Abadi and Cardelli's imperative object calculus, with simple and recursive types. Its operational and denotational semantics are given and related by an adequacy result. However, the remainder of the thesis relies on neither operational semantics nor the adequacy result.

The two main parts of the thesis are largely independent of each other. **Part II** is about Abadi and Leino's logic for the object calculus. In **Chapter 4** the logic is summarised, and a denotational semantics for its specifications is given. **Chapter 5** introduces the technically important notion of store specifications and contains the soundness proof for this logic. **Chapter 6** extends the logic and its soundness proof with recursive specifications. Finally, a discussion of limitations and possible further extensions of the logic in **Chapter 7** concludes this part.

Part III addresses the problem of finding a typed denotational model of the object calculus. **Chapter 8** introduces a higher-order calculus with general references and recalls a possible worlds model of this language due to Levy (2002, 2004). The extension with subtyping requires a coherence proof. Then, in **Chapter 9**, a semantics of objects is obtained by translation into the higher-order language. **Chapter 10** concludes with an outlook on local reasoning and a summary of further open questions.

1.7.3 Published Results

Some of the material presented in this thesis has been previously published. The results on which Part II is based were obtained as the result of joint research with my supervisor Bernhard Reus. A preliminary version of these results appeared in the following publications:

- Reus, B. and J. Schwinghammer (2006a). Denotational Semantics for a Program Logic of Objects. *Mathematical Structures in Computer Science 16*(2), 313–358.
- Reus, B. and J. Schwinghammer (2005). Denotational semantics for Abadi and Leino's logic of objects. In M. Sagiv (Ed.), *Proceedings of the European Symposium on Programming (ESOP '05)*, Volume 3444 of *Lecture Notes in Computer Science*, pp. 264–279. Springer.
- Reus, B. and J. Schwinghammer (2004). Denotational semantics for Abadi and Leino's logic of objects. Technical Report 2004:03, Informatics, University of Sussex.

In these articles we did not consider cloning, which required a minor modification in the semantics of objects. Part III is based on the following technical report:

- Schwinghammer, J. (2005a). A typed semantics for languages with higher-order store and subtyping. Technical Report 2005:05, Informatics, University of Sussex.

An extended abstract containing the coherence result appears as:

· Schwinghammer, J. (2005b). A typed semantics of higher-order store and subtyping. In M. Coppo et al. (Eds.), *Proceedings Ninth Italian Conference on Theoretical Computer Science (ICTCS '05)*, Volume 3701 of *Lecture Notes in Computer Science*, pp. 390–404. Springer.

Part I

Preliminaries

Chapter 2

Denotational Semantics

A *denotational semantics* explains the constructs of a programming language in terms of their *denotations* as elements of mathematical structures (Fiore et al. 1996). This is done in a compositional way, i.e., the meaning of a compound term is determined by the denotations of its constituents. Mathematical structures that have been used in denotational semantics include domains (certain, suitably complete, partially ordered sets) (Abramsky and Jung 1994; Mitchell 1996), games semantics (Abramsky and McCusker 1998), realizability structures and partial equivalence relations (Longley 1995), and functor categories (Tennent 1985).

In this thesis we use the term *domain* for partially ordered sets with least upper bounds for all countable chains, but not necessarily containing a least element (note that some authors call these structures *predomains*). Maps between domains are partial continuous functions, i.e., partial functions that are both monotone and preserve the least upper bounds. In the following sections we give precise definitions, recall some results from the literature, and fix our notation. We will need a few elementary notions from category theory, which can be found in (Pierce 1991) for instance.

There is no original material in this chapter. In particular, the generalisation of Pitts' work on relational structures on domains to relational structures on bilimit-compact categories is outlined in the book (Levy 2004). In the next section, the categories **Cpo** and **pCpo** are defined, and various type constructors are given. Sections 2.2, 2.3 and 2.4 summarise some key results about the solution of domain equations from work of Plotkin and Smyth (1982), Freyd (1991), and Levy (2004). Section 2.5 reviews the material of (Pitts 1996) that will be used in subsequent chapters.

2.1 Complete Partial Orders

A *partial order* $A = (A, \sqsubseteq_A)$ is a set A equipped with a reflexive, transitive, and antisymmetric relation $\sqsubseteq_A$. It is a *complete* partial order (cpo) if every ω-chain $a_0 \sqsubseteq_A a_1 \sqsubseteq_A a_2 \ldots$ in A has a least upper bound $\bigsqcup_{n\in\mathbb{N}} a_n$. If in addition A has a least element $\bot_A$, it is called a complete *pointed* partial order (cppo). If clear from context, we may drop the

subscript A.

For a partial map $f : A \rightharpoonup B$ and $a \in A$ we write $f(a)\downarrow$ if the application is defined, and $f(a)\uparrow$ otherwise. Composition $g \circ f$ (or just gf) of partial maps $f : A \rightharpoonup B$ and $g : B \rightharpoonup C$ is defined in the obvious way.

A partial map $f : A \rightharpoonup B$ between cpos A and B is *monotonic* iff for all $a_0, a_1 \in A$, whenever $a_0 \sqsubseteq a_1$ and $f(a_0)\downarrow$ then $f(a_1)\downarrow$ and $f(a_0) \sqsubseteq f(a_1)$. It is continuous if it preserves least upper bounds of ω-chains. More precisely, for monotonic $f : A \rightharpoonup B$ and an ω-chain $a_0 \sqsubseteq a_1 \sqsubseteq \ldots$ in A, let $\bigsqcup_n f(a_n)$ be undefined iff $f(a_n)\uparrow$ for all $n \in \mathbb{N}$, and otherwise denote the least upper bound of the chain $f(a_{n_0}) \sqsubseteq f(a_{n_0+1}) \sqsubseteq \ldots$ in B, where $n_0 \in \mathbb{N}$ is such that $f(a_{n_0})\downarrow$. Then f is *continuous* iff it is monotonic and for each ω-chain a it is the case that if either of $f(\bigsqcup_n a_n)$ and $\bigsqcup_n f(a_n)$ is defined then so is the other and they are equal.

We write **pCpo** for the category of cpos and partial continuous maps as morphisms, and **Cpo** for the subcategory of cpos and total continuous maps. Throughout the thesis we use '$\stackrel{def}{=}$' and '$\stackrel{def}{\Longleftrightarrow}$' for definitional equality and equivalence, respectively, and we write '$\equiv$' for syntactic equality.

Discrete Cpos and Lifting

Each set A gives rise to a *discrete* cpo where the partial order is equality. Every cpo A can be *lifted* into a cppo

$$A_\perp \stackrel{def}{=} A \cup \{\perp\}$$

by adjoining an element $\perp \notin A$ and making it least in the order. Lifting determines a functor $(-)_\perp : \mathbf{pCpo} \to \mathbf{Cpo}$,

$$(f)_\perp(a) \stackrel{def}{=} \begin{cases} \perp & \text{if } a = \perp \text{ or } f(a)\uparrow \\ f(a) & \text{otherwise} \end{cases}$$

for **pCpo**-morphisms $f : A \rightharpoonup B$. A cppo obtained by lifting a discrete cpo is called *flat*.

Products

The *product* $A \times B$ of cpos A and B is given by the componentwise partial order $\sqsubseteq_{A\times B}$ on the cartesian product of the underlying sets, i.e., $\langle a_1, b_1\rangle \sqsubseteq_{A\times B} \langle a_2, b_2\rangle$ iff $a_1 \sqsubseteq_A a_2$ and $b_1 \sqsubseteq_B b_2$. This is indeed the categorical product in **Cpo**. Moreover, this construction gives rise to a functor on **pCpo**, sending a pair of partial maps $f : A_1 \rightharpoonup A_2$ and $g : B_1 \rightharpoonup B_2$ to the partial map $f \times g : A_1 \times B_1 \rightharpoonup A_2 \times B_2$ where

$$(f \times g)(a, b) \stackrel{def}{=} \begin{cases} \langle f(a), g(b)\rangle & \text{if } f(a)\downarrow \text{ and } g(b)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

The restriction of this operation to total morphisms yields a functor $\mathbf{Cpo} \times \mathbf{Cpo} \longrightarrow \mathbf{Cpo}$.

This definition generalises to products $\prod_{i\in I} A_i$ of countable families of cpos $(A_i)_{i\in I}$. Any one-point cpo is a terminal object in **Cpo**; the empty cpo is terminal in **pCpo**.

Sums

The *disjoint union* $A_1 + A_2$ of cpos A_1 and A_2 is the cpo with underlying set

$$A_1 \uplus A_2 \stackrel{def}{=} \{\mathsf{in}_i(a) \mid i = 1,2 \ \wedge \ a \in A_i\}$$

The partial order is defined by $x \sqsubseteq_{A_1+A_2} x'$ iff there are $i \in \{1,2\}$ and $a, a' \in A_i$ such that $x = \mathsf{in}_i(a)$, $x' = \mathsf{in}_i(a')$ and $a \sqsubseteq_{A_i} a'$. (This makes $A + B$ the coproduct of A and B in **Cpo**.) The construction gives rise to a functor $\mathbf{pCpo} \times \mathbf{pCpo} \longrightarrow \mathbf{pCpo}$ that acts on partial maps $f : A_1 \rightharpoonup B_1$ and $g : A_2 \rightharpoonup B_2$ by

$$(f + g)(x) \stackrel{def}{=} \begin{cases} \mathsf{in}_1(f(a)) & \text{if } x = \mathsf{in}_1(a) \text{ for some } a \in A_1 \\ \mathsf{in}_2(g(a)) & \text{if } x = \mathsf{in}_2(a) \text{ for some } a \in A_2 \end{cases}$$

As for products, the restriction to total maps induces a functor $\mathbf{Cpo} \times \mathbf{Cpo} \longrightarrow \mathbf{Cpo}$. Finally, everything generalises to the disjoint union $\sum_{i \in I} A_i$ of countable families $(A_i)_{i \in I}$; in this case we will often write $\langle i, a \rangle$ in place of $\mathsf{in}_i(a)$. The empty cpo is the initial object in **Cpo** and **pCpo**.

Function Spaces

The *(continuous) function space* $[A \to B]$ consists of the set $\mathbf{Cpo}(A, B)$ of total continuous maps $A \to B$. It becomes a cpo when ordered pointwise:

$$f \sqsubseteq_{[A \to B]} g \quad \stackrel{def}{\iff} \quad \forall a \in A.\, f(a) \sqsubseteq_B g(a)$$

Thus if B is pointed then so is $[A \to B]$. The function space is the exponential in **Cpo**, so in particular **Cpo** is cartesian closed.

The function space determines a bifunctor $\mathbf{Cpo}^{op} \times \mathbf{Cpo} \longrightarrow \mathbf{Cpo}$, sending a $\mathbf{Cpo}^{op}$-morphism $f : A \to A'$ and a **Cpo**-morphism $g : B \to B'$ to

$$[f \to g](h) \stackrel{def}{=} ghf$$

for elements $h \in [A \to B]$.

The *partial function space* $[A \rightharpoonup B]$ is the cppo $\mathbf{pCpo}(A, B)$ of partial continuous maps from A to B ordered pointwise,

$$f \sqsubseteq_{[A \rightharpoonup B]} g \quad \stackrel{def}{\iff} \quad \forall a \in A.\, f(a)\!\downarrow \implies g(a)\!\downarrow \ \wedge f(a) \sqsubseteq_B g(a)$$

The least element is given by the function $\perp$ that is undefined on every $a \in A$. Finally, the partial function space construction can be turned into a bifunctor $\mathbf{pCpo}^{op} \times \mathbf{pCpo} \longrightarrow \mathbf{Cpo}$ by $[f \rightharpoonup g](h) = ghf$ for $f : A' \rightharpoonup A$, $g : B \rightharpoonup B'$ and all $h \in [A \rightharpoonup B]$.

Remark 2.1.1. *Note that, for every cpo A, $[A \rightharpoonup -] : \mathbf{pCpo} \longrightarrow \mathbf{Cpo}$ is a right adjoint for $- \times A : \mathbf{Cpo} \longrightarrow \mathbf{pCpo}$. Thus* **pCpo** *is* partial cartesian closed *(Fiore 1996).*

Records

For a set L let $\mathcal{P}_{\mathsf{fin}}(L)$ denote the set of its finite subsets. Let $\mathbb{L}$ be a countable set and A a cpo. The cpo of *records with labels from $\mathbb{L}$ and entries in A* is defined as follows:

$$\mathsf{Rec}_{\mathbb{L}}(A) \stackrel{def}{=} \textstyle\sum_{L \in \mathcal{P}_{\mathsf{fin}}(\mathbb{L})} A^L$$

Table 2.1 *Update and extension of records*

$$\{\!|l_i = a_i|\!\}_{i=1\ldots n}[l := a] \stackrel{def}{=} \begin{cases} \{\!|l_1 = a_1, \ldots, l_k = a, \ldots, l_n = a_n|\!\} & \text{if } l = l_k \text{ for some } k \in \{1,\ldots,n\} \\ \{\!|l_1 = a_1, \ldots, l_n = a_n, l = a|\!\} & \text{otherwise} \end{cases}$$

Table 2.2 *Composition of records*

$$\{\!|l_i = a_i|\!\}_{i\in I} + \{\!|l_j = b_j|\!\}_{j\in J} \stackrel{def}{=} \begin{cases} \{\!|l_i = a_i, l_j = b_j|\!\}_{i\in I, j\in J} & \text{if } \{l_i \mid i \in I\} \cap \{l_j \mid j \in J\} = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

where A^L is the set of all total functions from L to A. Viewing $\mathbb{L}$ as a flat cpo the ordering on $\mathsf{Rec}_{\mathbb{L}}(A)$ is determined as

$$r_1 \sqsubseteq_{\mathsf{Rec}_{\mathbb{L}}(A)} r_2 \iff r_i = \langle L_i, f_i\rangle \Longrightarrow L_1 = L_2 \ \wedge\ \forall l \in L_1.\, f_1(l) \sqsubseteq_A f_2(l) \tag{2.1}$$

Note that $\mathsf{Rec}_{\mathbb{L}}(A)$ is always non-empty since it contains $\langle \emptyset, \epsilon\rangle$ for $\epsilon : \emptyset \to A$ the empty map. $\mathsf{Rec}_{\mathbb{L}}$ extends to an endofunctor $\mathsf{Rec}_{\mathbb{L}}(-)$ on **pCpo**, sending $g : A \to B$ to the partial map $\mathsf{Rec}_{\mathbb{L}}(g) : \mathsf{Rec}_{\mathbb{L}}(A) \to \mathsf{Rec}_{\mathbb{L}}(B)$ defined by

$$\mathsf{Rec}_{\mathbb{L}}(g)(\langle L, f\rangle) = \begin{cases} \langle L, gf\rangle & \text{if } g(f(l))\downarrow \text{ for all } l \in L \\ \text{undefined} & \text{otherwise} \end{cases}$$

A record $r = \langle L, f\rangle \in \mathsf{Rec}_{\mathbb{L}}(A)$, with labels $L = \{l_1, \ldots, l_n\}$ and corresponding entries $f(l_i) = a_i$, is written as $\{\!|l_1 = a_1, \ldots, l_n = a_n|\!\}$. Moreover, for such records we also define $\mathsf{dom}(r) \stackrel{def}{=} L$.

Update and extension of records is defined in Table 2.1 by the corresponding operation on functions; Table 2.2 defines a partial composition operation. These operations are continuous. From (2.1) it follows that a record and a proper extension are incomparable. Selection of a component labelled $l \in \mathbb{L}$ of a record $r \in \mathsf{Rec}_{\mathbb{L}}(A)$ is written $r.l$. It is defined and yields $f(l) \in A$ if r is $\langle L, f\rangle$ and $l \in L$; it is undefined if $l \notin L$.

Typed Records

Suppose $\mathbb{L}$ is a fixed, countable set of labels. Given a finite set I, cpos A_i and labels $l_i \in \mathbb{L}$ for all $i \in I$, the cpo of *typed records, with entries labelled l_i in A_i for $i \in I$*, is given by the set

$$\{\!|l_i : A_i|\!\}_{i\in I} \stackrel{def}{=} \{r : \{l_i | i \in I\} \to \textstyle\bigcup_{i\in I} A_i \,\big|\, \forall i \in I.\, r(l_i) \in A_i\}$$

ordered pointwise: $r \sqsubseteq_{\{\!|l_i:A_i|\!\}} r'$ iff $r(l_i) \sqsubseteq_{A_i} r'(l_i)$ for all $i \in I$. As in the case of the (untyped) records above we write $r.l$ for $r(l)$, and $\{\!|l_i = a_i|\!\}_{i\in I}$ for the map $r \in \{\!|l_i : A_i|\!\}_{i\in I}$ such that $r(l_i) = a_i \in A_i$ for all $i \in I$. If clear from context we may omit the index set.

This construction is functorial, sending partial continuous maps $f_i : A_i \to B_i$ to the partial map $\{\!|l_i : f_i|\!\} : \{\!|l_i : A_i|\!\} \to \{\!|l_i : B_i|\!\}$,

$$\{\!|l_i : f_i|\!\}_{i\in I}(r) \stackrel{def}{=} \begin{cases} \{\!|l_i = f_i(r.l_i)|\!\}_{i\in I} & \text{if } f_i(r.l_i)\downarrow \quad \forall i \in I \\ \text{undefined} & \text{otherwise} \end{cases}$$

Obviously the restriction to total continuous maps determines a functor $\mathbf{Cpo}^I \longrightarrow \mathbf{Cpo}$. Update and composition can be defined as in Tables 2.1 and 2.2.

Least Fixed Points

Suppose A is a cppo, and $f : A \to A$ is a total continuous map. Then f has a least (pre-) fixed point $lfp(f) \in A$, given by the least upper bound of the ω-chain $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$ in A:

$$f(lfp(f)) = lfp(f) \quad \wedge \quad (\forall a \in A. f(a) \sqsubseteq a \implies lfp(f) \sqsubseteq a)$$

This observation gives rise to the principle of *fixed point induction*: Suppose $P \subseteq A$ contains $\bot$ and is closed under taking least under bounds of ω-chains in P. If P is also closed under application of f, then $lfp(f) \in P$.

Least fixed points are uniform, in the following sense (Plotkin 1983; Abramsky and Jung 1994): Let B be another cppo, let $g : B \to B$ be total and continuous, and suppose $\epsilon : A \to B$ is a total continuous map that also preserves the least element and such that $g\epsilon = \epsilon f$. Then $lfp(g) = \epsilon(lfp(f))$.

2.2 Bilimit-Compact Categories

The notion of bilimit-compactness (Levy 2004) abstracts a sufficient condition for the existence of solutions to recursive domain equations. It turns out that bilimit-compact categories are closed under the operations of constructing the opposite category, products and functor categories. In this section the basic definitions and constructions are given; the construction of (minimal) "invariant" objects is considered in the next section.

Recall that a **Cpo**-enriched category is a category C where

- for every pair of C-objects A and B, the hom-set $C(A, B)$ is equipped with a complete partial order $\sqsubseteq_{C(A,B)}$;
- composition of C-arrows is continuous with respect to these orderings: if $f \sqsubseteq f'$ in $C(A, B)$ and $g \sqsubseteq g'$ in $C(B, C)$, then $gf \sqsubseteq g'f'$ in $C(A, C)$, and if $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ and $g_0 \sqsubseteq g_1 \sqsubseteq \ldots$ are ω-chains in $C(A, B)$ and $C(B, C)$, resp., then $\bigsqcup_n (g_n f_n) = (\bigsqcup_n g_n)(\bigsqcup_n f_n)$.

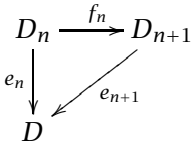
Both **Cpo** and **pCpo** are **Cpo**-enriched. If C is **Cpo**-enriched then so is the opposite category C^{op}, with the order $f \sqsubseteq_{C^{op}(A,B)} g$ iff $f \sqsubseteq_{C(B,A)} g$. Any small product $\prod_{i \in I} C_i$ of **Cpo**-enriched categories C_i is **Cpo**-enriched, with hom-sets ordered componentwise. Details can be found in (Levy 2004).

A functor $F : \mathcal{B} \longrightarrow C$ between **Cpo**-enriched categories $\mathcal{B}$ and C is *locally continuous* if and only if it is continuous on the hom-sets, that is, if $\bigsqcup_n F(f_n) = F(\bigsqcup_n f_n)$ holds for all ω-chains $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ in $\mathcal{B}(A, B)$. All the (bi-)functors given in Section 2.1 are locally continuous. Moreover, composition of functors preserves local continuity.

Recall that a C-morphism $e : A \to B$ is an *embedding* if there is a corresponding *projection* morphism $p : B \to A$, i.e., such that $pe = \mathsf{id}_A$ and $ep \sqsubseteq \mathsf{id}_B$. Embeddings and projections uniquely determine each other, and we write e° for the projection corresponding to an embedding e. Note that if $F : C^{op} \times C \longrightarrow C$ is a locally continuous bifunctor and

$e : A \to B$ is an embedding, then $F(e^\circ, e) : F(A,A) \to F(B,B)$ is an embedding where $F(e^\circ, e)^\circ = F(e, e^\circ)$.

Let $\Delta = D_0 \xrightarrow{f_0} D_1 \xrightarrow{f_1} \ldots$ be an ω-chain in the category C^E of embeddings of C. A *cocone* $(e_n)_{n\in\mathbb{N}} : \Delta \to D$ of Δ consists of a C-object D together with embeddings e_n from D_n to D, such that, for all $n \in \mathbb{N}$,



commutes in C^E. Such a cocone is a *bilimit* for Δ iff $\bigsqcup_n (e_n\, e_n^\circ) = \mathsf{id}_D$ in $C(D,D)$ (note that $e_n\, e_n^\circ \sqsubseteq e_{n+1}\, e_{n+1}^\circ$, so the least upper bound does exist). It follows that any bilimit $(e_n)_{n\in\mathbb{N}} : \Delta \to D$ is a colimit for Δ, for if $(i_n)_{n\in\mathbb{N}} : \Delta \to D'$ is a cocone then the unique mediating morphism $D \to D'$ is given by the embedding $\bigsqcup_n (i_n\, e_n^\circ)$.

Definition 2.2.1 (Levy 2004)**.** *A* **Cpo***-enriched category C is* bilimit-compact *if*

- *each hom-cpo $C(A,B)$ has a least element $\bot$;*
- *composition is bi-strict, i.e., $\bot \circ f = \bot = g \circ \bot$;*
- *C has an initial object (which, by bi-strictness, is also terminal);*
- *every ω-chain in the category C^E of embeddings of C has a bilimit.*

That the category **pCpo** is bilimit-compact is well-known; a bilimit of an ω-chain of embeddings $\Delta = D_0 \xrightarrow{f_0} D_1 \xrightarrow{f_1} \ldots$ is given by the cpo

$$D \stackrel{def}{=} \left\{ x \in \prod_{n\in\mathbb{N}} D_n \;\middle|\; \forall n \in \mathbb{N}.\, x_n = f_n^\circ(x_{n+1}) \right\}$$

partially ordered componentwise, and the embeddings $e_n : D_n \to D$ are determined by the projections $x \mapsto x_n$ from D to D_n.

Further, if C is bilimit-compact then so is C^{op}, and any small product $\prod_{i\in I} C_i$ of bilimit-compact categories C_i is bilimit-compact.

2.3 Minimal Invariants

Let $F : C^{op} \times C \longrightarrow C$ be a locally continuous bifunctor on a **Cpo**-enriched category C. An *invariant* for F is an object D of C together with an isomorphism $i : F(D,D) \cong D$. It is a *minimal* invariant for F if the identity map id_D on D is the least fixed point of the continuous endofunction $\delta(e) \stackrel{def}{=} iF(e,e)i^{-1}$ on $C(D,D)$.

The existence of solutions to recursive domain equations $\alpha = \Phi(\alpha)$, where Φ contains (some of) the domain constructors of Section 2.1, can be reduced to finding invariant objects: As mentioned above, the constructions on domains give rise to a (bi)-functor $F(D^-, D^+)$, obtained from Φ by separating positive and negative occurrences of α in Φ. Obviously, by diagonalisation, an invariant $i : F(D,D) \cong D$ is a fixed point for Φ (up to isomorphism). As discussed below, *minimal* invariants $i : F(D,D) \cong D$ for F are uniquely determined (if they exist). Thus we can write $rec\,\alpha.\Phi(\alpha)$ for this domain D.

2.3.1 Uniqueness of Minimal Invariants

Freyd (1991) characterised minimal invariants by the following universal property. It states that, if D is a minimal invariant for F, then the object $\langle D, D\rangle$ of $C^{op} \times C$ is simultaneously initial in the category of $\hat{F}$-algebras, and terminal in the category of $\hat{F}$-coalgebras, for the functor $\hat{F}(X,Y) \stackrel{def}{=} \langle F(Y,X), F(X,Y)\rangle : C^{op} \times C \longrightarrow C^{op} \times C$. Here we consider this for the case of bilimit-compact categories only, which suffices for the applications in this thesis.

Theorem 2.3.1 (Freyd 1991; Pitts 1996)**.** *Suppose $F : C^{op} \times C \longrightarrow C$ is a locally continuous bifunctor on a bilimit-compact category C. Further, suppose $i : F(D,D) \cong D$ is a minimal invariant for F. For each pair of objects A, B and pair of morphisms $f : A \to F(B,A)$ and $g : F(A,B) \to B$ of C, there exist unique C-morphisms $h : A \to D$ and $k : D \to B$ such that the squares*

$$\begin{array}{ccc} D & \xrightarrow{i^{-1}} & F(D,D) \\ {\scriptstyle h}\uparrow & & \uparrow{\scriptstyle F(k,h)} \\ A & \xrightarrow[f]{} & F(B,A) \end{array} \qquad \begin{array}{ccc} F(D,D) & \xrightarrow{i} & D \\ {\scriptstyle F(h,k)}\downarrow & & \downarrow{\scriptstyle k} \\ F(A,B) & \xrightarrow[g]{} & B \end{array} \tag{2.2}$$

commute. In particular, any two minimal invariants for F are isomorphic.

Sketch of Proof. Suppose h, k is any pair of morphisms for which the diagrams (2.2) commute. Consider the continuous map $\epsilon : C(D,D) \to C(A,D) \times C(D,B)$ defined by $\epsilon(e) \stackrel{def}{=} \langle eh, ke\rangle$. If $\delta(e) = iF(e,e)i^{-1}$ as above, then by the assumptions on h and k and the functoriality of F,

$$\begin{aligned} \epsilon(\delta(e)) &= \langle iF(e,e)i^{-1}h, kiF(e,e)i^{-1}\rangle \\ &= \langle iF(ke,eh)f, gF(eh,ke)i^{-1}\rangle \end{aligned}$$

Thus the following diagram commutes

$$\begin{array}{ccc} C(D,D) & \xrightarrow{\epsilon} & C(A,D) \times C(D,B) \\ {\scriptstyle \delta}\downarrow & & \downarrow{\scriptstyle \phi} \\ C(D,D) & \xrightarrow[\epsilon]{} & C(A,D) \times C(D,B) \end{array}$$

where $\phi(h,k) \stackrel{def}{=} \langle iF(k,h)f, gF(h,k)i^{-1}\rangle$. By bistrictness of composition in C, ϵ is strict, and $lfp(\phi) = \epsilon(lfp(\delta))$ follows from the uniformity of least fixed points. By minimality of D, $lfp(\delta) = \mathsf{id}_D$. Hence, $lfp(\phi) = \epsilon(\mathsf{id}_D) = \langle h, k\rangle$ is the unique pair of maps satisfying the property (2.2).

If E is another minimal invariant for F then, taking A and B in (2.2) to be E, the maps h and k establish the isomorphism $k : D \cong E$. □

2.3.2 Existence of Minimal invariants

Theorem 2.3.2 (Levy 2004; Pitts 1996)**.** *Suppose C is bilimit-compact, and $F : C^{op} \times C \longrightarrow C$ is locally continuous. Then F has a minimal invariant.*

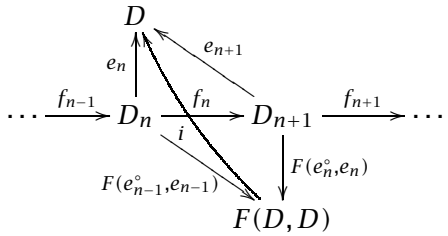
Sketch of Proof. Bilimit-compactness of C entails that the unique maps out of and into the initial object form embedding-projection pairs. By local continuity of F we can therefore construct an ω-chain Δ in C^E,

$$\Delta \stackrel{def}{=} D_0 \xrightarrow{f_0} D_1 \xrightarrow{f_1} \dots$$

where D_0 is the initial object of C, $D_{n+1} = F(D_n, D_n)$, f_0 is the unique morphism $D_0 \to D_1$ in C, and $f_{n+1} = F(f_n^\circ, f_n)$ is an embedding. Let $(e_n)_{n\in\mathbb{N}} : \Delta \to D$ be the bilimit for Δ. It is easy to check that $(F(e_n^\circ, e_n))_{n\in\mathbb{N}} : F(\Delta, \Delta) \to F(D, D)$ is a bilimit for the chain

$$F(\Delta,\Delta) = F(D_0, D_0) \xrightarrow{F(f_0^\circ, f_0)} F(D_1, D_1) \xrightarrow{F(f_1^\circ, f_1)} \dots$$

But this is just the chain $\Delta^- \stackrel{def}{=} D_1 \xrightarrow{f_1} D_2 \xrightarrow{f_2} \dots$ whose bilimit is again D. Since both D and $F(D, D)$ are colimits for Δ^- there is a unique isomorphism $i : F(D, D) \cong D$, such that $iF(e_n^\circ, e_n) = e_{n+1}$ for all $n \in \mathbb{N}$,



Now $e_0\, e_0^\circ = \bot_{C(D,D)}$, by bi-strictness and the fact that $e_0 = \bot_{C(D_0,D)}$ is uniquely determined by initiality. By induction on n we obtain $e_{n+1}\, e_{n+1}^\circ = iF(e_n\, e_n^\circ, e_n\, e_n^\circ)i^{-1} = \delta^{n+1}(\bot)$ for $\delta(e) = iF(e, e)i^{-1}$. By the defining property of bilimits, $lfp(\delta) = \bigsqcup_n \delta^n(\bot) = \bigsqcup_n (e_n\, e_n^\circ) = \mathsf{id}_D$, i.e., $i : F(D, D) \cong D$ is a minimal invariant for F. □

Lemma 2.3.3. *With notation as in the proof of Theorem 2.3.2 above, $\delta^n(\bot) \circ \delta^m(\bot) = \delta^n(\bot)$ for all $n, m \in \mathbb{N}$ with $m \geq n$.*

Proof. Observe that for $m \geq n$ we have $e_n = e_m f_{m-1} \cdots f_n$, hence

$$\begin{aligned}\delta^n(\bot)\delta^m(\bot) &= e_n(f_n^\circ \cdots f_{m-1}^\circ e_m^\circ)e_m e_m^\circ \\ &= e_n(f_n^\circ \cdots f_{m-1}^\circ)e_m^\circ \\ &= e_n e_n^\circ = \delta^n(\bot)\end{aligned}$$

□

2.3.3 Sub-Bilimit-Compact Categories

As discussed in (Levy 2004), one often wants to solve domain equations in categories which are not bilimit-compact. For instance, while computations in a call-by-value language usually denote morphisms in **pCpo**, conceptually, values $\Gamma \rhd v : A$ of (possibly recursively defined) types A should be interpreted by morphisms from Γ to A in **Cpo** which fails to be bilimit-compact. However, **Cpo** is a subcategory of the bilimit-compact category **pCpo** such that all isomorphisms of **pCpo** are in fact already morphisms of **Cpo**.

This entails that minimal invariants can be found in **Cpo**. For the general case, the notion of *sub-bilimit-compactness* of (Levy 2004) ensures the property that all isomorphisms of a bilimit-compact supercategory are available.

Recall that a subcategory $\mathcal{B}$ of $\mathcal{C}$ is *full* if $\mathcal{B}(A,B) = \mathcal{C}(A,B)$ for all $\mathcal{B}$-objects A and B. It is *lluf*, or a *wide subcategory*, if every object of $\mathcal{C}$ is an object of $\mathcal{B}$. A subcategory of a **Cpo**-enriched category $\mathcal{C}$ is *admissible* if it is closed under taking least upper bounds of ω-chains. For $\mathcal{B}$ a subcategory of $\mathcal{C}$ we write $\mathcal{B} \bullet\!\!\to \mathcal{C}$ for the unique category $\mathcal{D}$ such that $\mathcal{B} \subseteq_{lluf} \mathcal{D} \subseteq_{full} \mathcal{C}$, i.e., $\mathcal{B} \bullet\!\!\to \mathcal{C}$ has the same objects as $\mathcal{B}$, and all $\mathcal{C}$-morphisms between $\mathcal{B}$-objects.

Definition 2.3.4 (Levy 2004)**.** *Suppose $\mathcal{C}$ is bilimit-compact. We say a* **Cpo***-enriched category $\mathcal{B}$ is* sub-bilimit-compact *within $\mathcal{C}$ if the following conditions hold.*

- *$\mathcal{C}$ contains $\mathcal{B}$ as a lluf admissible subcategory.*
- *Suppose $\Delta = D_0 \to D_1 \to \ldots$ and $\Delta' = D'_0 \to D'_1 \to \ldots$ are ω-chains in $\mathcal{C}^E$ with bilimits $(e_n)_{n\in\mathbb{N}} : \Delta \to D$ and $(e'_n)_{n\in\mathbb{N}} : \Delta' \to D'$, resp. Further, suppose $\mu : \Delta \dot{\to} \Delta'$ is a natural transformation from Δ to Δ' such that $\mu_n \in \mathcal{B}(D_n, D'_n)$ for all n. Then the least upper bound $\bigsqcup_n (e'_n \mu_n e^\circ_n)$ is in $\mathcal{B}(D, D')$.*

It follows that a sub-bilimit-compact category $\mathcal{B}$ within $\mathcal{C}$ contains already all the isomorphisms $i : D \cong D'$ of $\mathcal{C}$ (Levy 2004): This can be seen by considering the constant ω-chains $\Delta = \Delta' = D \xrightarrow{\mathsf{id}} D \xrightarrow{\mathsf{id}} \ldots$, with bilimits $(\mathsf{id})_n : \Delta \to D$ and $(i)_n : \Delta' \to D'$, and $\mu = (\mathsf{id})_n : \Delta \dot{\to} \Delta'$. By definition of sub-bilimit-compactness, $i = \bigsqcup_n (i\, \mathsf{id}_A\, \mathsf{id}_A)$ is in $\mathcal{B}(D, D')$.

Any bilimit-compact category is sub-bilimit-compact within itself. Note that the embeddings e'_n in Definition 2.3.4 are necessarily total. Therefore it is easy to see that **Cpo** is sub-bilimit-compact within **pCpo**: By virtue of bilimit-compactness of **pCpo**, for all x and sufficiently large $n \in \mathbb{N}$, $e^\circ_n(x)\downarrow$, i.e. the least upper bound in Definition 2.3.4 is total and therefore a **Cpo**-morphism. Further, if $\mathcal{B}$ is sub-bilimit-compact within $\mathcal{C}$ then so is $\mathcal{B}^{op}$ within $\mathcal{C}^{op}$. If $\mathcal{B}_i$ is sub-bilimit-compact within $\mathcal{C}_i$ for all $i \in I$, then $\prod_{i\in I} \mathcal{B}_i$ is sub-bilimit-compact within $\prod_{i\in I} \mathcal{C}_i$.

2.4 Functor Categories

The possible-world models that we present later are based on functor categories. Models based on functor categories have proved successful in the treatment of the dynamically changing "store shapes" of both Algol-like languages with block-allocated memory (Oles 1982; Tennent 1985; O'Hearn and Tennent 1997) and languages with more general dynamic allocation (for instance, Levy 2002; Stark 1996).

For our purpose it suffices to consider categories of locally continuous functors between **Cpo**-enriched categories: If $\mathcal{I}$ and $\mathcal{C}$ are **Cpo**-enriched, then $[\mathcal{I}, \mathcal{C}]$ denotes the category of locally continuous functors $\mathcal{I} \longrightarrow \mathcal{C}$ and natural transformations between them. Further, $[\mathcal{I}, \mathcal{C}]$ can be shown to be **Cpo**-enriched when natural transformations $F \dot{\to} G$ are ordered pointwise, $\mu \sqsubseteq_{[\mathcal{I},\mathcal{C}](F,G)} \nu$ iff $\mu_A \sqsubseteq_{\mathcal{C}(FA,GA)} \nu_A$ for all $\mathcal{I}$-objects A. In particular, a

natural transformation $\mu : F \dot{\to} G$ is an embedding iff each component μ_A is an embedding $FA \to GA$ in $\mathcal{C}$.

Proposition 2.4.1 (Levy 2004)**.** *If $\mathcal{I}$ is a (small)* **Cpo***-enriched category and $\mathcal{C}$ is bilimit-compact then so is the category $[\mathcal{I}, \mathcal{C}]$. Moreover, if a* **Cpo***-enriched category $\mathcal{B}$ is sub-bilimit-compact within $\mathcal{C}$, then $[\mathcal{I}, \mathcal{B}]$ is sub-bilimit-compact within $[\mathcal{I}, \mathcal{B}] \bullet\!\!\to [\mathcal{I}, \mathcal{C}]$.*

Proof. For the first part, observe that the natural transformation $\mu : F \dot{\to} G$ given at A by $\mu_A = \bot_{\mathcal{C}(FA,GA)}$ is least in $[\mathcal{I}, \mathcal{C}](F, G)$; bi-strictness of composition in $[\mathcal{I}, \mathcal{C}]$ is inherited from $\mathcal{C}$. The initial object is given by the constant functor 0, where 0 is the initial object of the category $\mathcal{C}$.

Finally, bilimits are computed pointwise: An ω-chain $\Phi = F_0 \xrightarrow{\mu_0} F_1 \xrightarrow{\mu_1} \ldots$ in $[\mathcal{I}, \mathcal{C}]^E$ gives rise to an ω-chain $\Phi A = F_0 A \xrightarrow{\mu_{0A}} F_1 A \xrightarrow{\mu_{1A}} \ldots$ in $\mathcal{C}^E$, for each $\mathcal{I}$-object A, with bilimit $(e_{An})_{n\in\mathbb{N}} : \Phi A \to F_A$. Let F be the (locally continuous) functor defined by $FA = F_A$ on $\mathcal{I}$-objects A, and $Ff = \bigsqcup_n (e_{Bn} \, (F_n f) \, e^\circ_{An})$ on $\mathcal{I}$-morphisms $f : A \to B$. If ν_n denotes the natural transformation $F_n \dot{\to} F$ given at A by e_{An}, then $(\nu_n)_{n\in\mathbb{N}} : \Phi \to F$ is a cocone. Since $(\bigsqcup_n (\nu_n \, \nu_n^\circ))_A = \bigsqcup_n (e_{An} \, e^\circ_{An}) = \mathsf{id}_{FA}$ for each A, i.e., $\bigsqcup_n (\nu_n \, \nu_n^\circ) = \mathsf{id}_F$, it follows that F is a bilimit for Φ.

For the second claim, let $F_0 \xrightarrow{\mu_0} F_1 \xrightarrow{\mu_1} \ldots$ an ω-chain in $[\mathcal{I}, \mathcal{B}] \bullet\!\!\to [\mathcal{I}, \mathcal{C}]$. Recall that the objects of $[\mathcal{I}, \mathcal{B}] \bullet\!\!\to [\mathcal{I}, \mathcal{C}]$ are functors $F, G \in [\mathcal{I}, \mathcal{B}]$ and morphisms are natural transformations $\mu : F \dot{\to} G$ such that for all $A \in \mathcal{I}$, $\mu_A \in \mathcal{C}(FA, GA)$. As seen in the first part of this proof, bilimits in $[\mathcal{I}, \mathcal{C}]$ are given by natural transformations $(\nu_n)_{n\in\mathbb{N}}$ where $\nu_{nA} = e_{An} \in \mathcal{C}^E(F_n A, F_A)$. By assumption, $\mathcal{B}$ is sub-bilimit-compact within $\mathcal{C}$, hence F_A is already in $\mathcal{B}$ and $[\mathcal{I}, \mathcal{B}] \bullet\!\!\to [\mathcal{I}, \mathcal{C}]$ is bilimit-compact. By definition, $[\mathcal{I}, \mathcal{B}]$ is a lluf subcategory; admissibility follows easily from the pointwise ordering of natural transformations and the assumption that $\mathcal{B}$ is sub-bilimit-compact within in $\mathcal{C}$. Similarly, the required property about natural transformations $\mu : F \dot{\to} F'$ between bilimits in Definition 2.3.4 follows from the pointwise construction of bilimits above and the assumption that $\mathcal{B}$ is sub-bilimit-compact within $\mathcal{C}$. □

Note that Proposition 2.4.1 applies in particular to any (small) category $\mathcal{I}$ when the hom-sets are regarded as discretely ordered cpos.

2.5 Relational Properties of Bilimit-Compact Categories

When reasoning about elements of a recursively defined domain $D = rec\alpha.\Phi(\alpha)$, one often looks for an *invariant* relation $\Delta = \Phi(\Delta)$ on D. Establishing the existence of a specific instance of such relations can usually be done by considering the detailed construction of $rec\alpha.\Phi(\alpha)$. Examples of this method can be found, e.g., in (Reynolds 1974) and (Kamin and Reddy 1994) where correspondence ('adequacy') between two denotational semantics (involving recursively defined domains) are proved.

In contrast, Pitts (1996) provides a general framework to find such invariant relations. The method relies on extending the functor $F(D^-, D^+)$ corresponding to the domain $D = rec\alpha.\Phi(\alpha)$ with an action on *relations*, for a general notion of relation. While Pitts uses

the category of cppos and $\perp$-preserving continuous maps, Levy (2004) pointed out that everything generalises to arbitrary bilimit-compact categories. We present the material for this general case.

2.5.1 Relational Structures

A *(normal) relational structure* $\mathcal{R}$ on a category C consists of (1) a set of $\mathcal{R}$*-relations on* D, $\mathcal{R}(D)$, for each C-object D, and (2) a binary relation $f : - \subset -$ between elements of $\mathcal{R}(D)$ and elements of $\mathcal{R}(E)$, for each C-morphism $f : D \to E$, and such that the following conditions are satisfied:

(Identity) $\mathsf{id}_D : R \subset R$, for all $R \in \mathcal{R}(D)$;

(Composition) $gf : R \subset T$, for all composable $f : R \subset S$ and $g : S \subset T$;

(Normality) if $\mathsf{id}_D : R \subset S$ and $\mathsf{id}_D : S \subset R$, then $R = S$.

As explained in (Pitts 1996), informally, $f : R \subset S$ means that f maps elements 'related' by R to elements 'related' by S. For example, let $\mathcal{W} = (\mathcal{W}, \leq)$ be a poset considered as a category, and let C be the functor category $[\mathcal{W}, \mathbf{pCpo}]$. A relational structure $\mathcal{K}$ of "Kripke relations" on C is given if

- $\mathcal{K}(A)$ consists of all families $R = (R_w)_{w \in \mathcal{W}}$ of subsets $R_w \subseteq A(w)$ that are closed under taking least upper bounds, and such that $x \in R_w$ implies $A(w \leq w')(x) \in R_{w'}$ whenever $A(w \leq w')(x)\downarrow$; and
- $\mu : R \subset S$ if and only if, for all $w \in \mathcal{W}$ and for all $x \in R_w$, $\mu_w(x) \in S_w$ whenever $\mu_w(x)\downarrow$, for all $R = (R_w)_w \in \mathcal{K}(A)$ and $S = (S_w)_w \in \mathcal{K}(B)$

where A, B are C-objects, and $\mu : A \stackrel{\cdot}{\to} B$ is a natural transformation. Similar relational structures will be considered in the following chapters.

In the remainder of this section let C be an arbitrary, bilimit-compact category. If $\mathcal{R}$ is a relational structure on C, an $\mathcal{R}$-relation $S \in \mathcal{R}(E)$ is *admissible* iff, for all $\mathcal{R}$-relations $R \in \mathcal{R}(D)$, the subset

$$[R, S] \stackrel{def}{=} \{f \mid f : R \subset S\}$$

of the hom-cpo $C(D, E)$ contains $\perp$ and is closed under taking least upper bounds of ω-chains. For C-objects D let $\mathcal{R}_{adm}(D)$ denote the subset of admissible $\mathcal{R}$-relations on D. For instance, for the Kripke relations defined above we have $\mathcal{K}_{adm}(A) = \mathcal{K}(A)$ for all A.

2.5.2 Invariant $\mathcal{R}$-relations

An *admissible action* of a functor $F : (C^{op})^m \times C^n \longrightarrow C$ (where $m, n \geq 0$) on $\mathcal{R}$-relations is an operation Φ mapping (tuples of) $\mathcal{R}$-relations $R = (R_i)_{i=1}^m \in \prod_{i=1}^m \mathcal{R}(D_i)$ and $S = (S_j)_{j=1}^n \in \prod_{j=1}^n \mathcal{R}(E)$ to an $\mathcal{R}$-relation $\Phi(R, S) \in \mathcal{R}(F(D_1, \ldots, D_m, E_1, \ldots, E_n))$ such that the following conditions hold:

- if each S_j is admissible then, for any R, also $\Phi(R, S)$ is admissible;

- $F(f_1,\ldots,f_m,g_1,\ldots,g_n) : \Phi(R,S) \subset \Phi(R',S')$ whenever $f_i : R'_i \subset R_i$ and $g_j : S_j \subset S'_j$ for all i and j.

In general, there may be many admissible actions for a given functor F and relational structure $\mathcal{R}$. Pitts (1996) provides several examples of admissible actions for functors on the category **Cpo**.

Now suppose that a locally continuous functor $F : C^{op} \times C \longrightarrow C$ is equipped with an admissible action Φ on $\mathcal{R}$-relations. By Theorem 2.3.2 there exists a minimal invariant $i : F(D,D) \cong D$ for F. We call an admissible $\mathcal{R}$-relation $\Delta \in \mathcal{R}_{adm}(D)$ an *invariant $\mathcal{R}$-relation for F* if and only if $i : \Phi(\Delta,\Delta) \subset \Delta$ and $i^{-1} : \Delta \subset \Phi(\Delta,\Delta)$.

2.5.3 Uniqueness of Invariant $\mathcal{R}$-relations

Invariant $\mathcal{R}$-relation for locally continuous functors are unique if they exist. This will follow from the analogue of Theorem 2.3.1 for relational structures, given next.

Proposition 2.5.1 (Pitts 1996)**.** *Let $\mathcal{R}$ be a relational structure on a bilimit-compact category C. Let $F : C^{op} \times C \longrightarrow C$ be a locally continuous functor with an admissible action Φ on $\mathcal{R}$, and let $i : F(D,D) \cong D$ be its minimal invariant. Suppose Δ is a minimal invariant $\mathcal{R}$-relation for F.*

For any C-morphisms $f : A \to F(B,A)$ and $g : F(A,B) \to B$, and any relations $R \in \mathcal{R}(A)$ and $S \in \mathcal{R}_{adm}(B)$, if $f : R \subset \Phi(S,R)$ and $g : \Phi(R,S) \subset S$, then

$$h : R \subset \Delta \quad \textit{and} \quad k : \Delta \subset S$$

where $\langle h,k\rangle \stackrel{def}{=} lfp(\phi)$ are the unique C-morphisms determined by f and g as in Theorem 2.3.1.

Sketch of Proof. This is proved exactly as in (Pitts 1996), relying on the proof of Theorem 2.3.1: For any $h' \in C(A,D)$, $k' \in C(D,B)$, if $h' : R \subset \Delta$ and $k' : \Delta \subset S$, then

$$F(k',h') : \Phi(S,R) \subset \Phi(\Delta,\Delta) \quad \text{and} \quad F(h',k') : \Phi(\Delta,\Delta) \subset \Phi(R,S)$$

by admissibility of the action of F. This entails $iF(k',h')f : R \subset \Delta$ and $gF(h',k')i^{-1} : \Delta \subset S$. Hence, the function $\phi(h',k') = \langle iF(k',h')f, gF(h',k')i^{-1}\rangle$ from Theorem 2.3.1 maps the set

$$[R,\Delta] \times [\Delta,S] = \{\langle h,k\rangle \mid h : R \subset \Delta \ \wedge \ k : \Delta \subset S\}$$

into itself. Both Δ and S are admissible $\mathcal{R}$-relations, therefore $[R,\Delta] \times [\Delta,S]$ is a chain-closed subset of $C(A,D)\times C(D,B)$ containing $\langle \bot,\bot\rangle$. By fixed point induction, also $\langle h,k\rangle = lfp(\phi) \in [R,\Delta] \times [\Delta,S]$ which proves the proposition. □

Choosing $A = B = D$, it follows that for all $R \in \mathcal{R}(D)$, $S \in \mathcal{R}_{adm}(D)$, if $i^{-1} : R \subset \Phi(S,R)$ and $i : \Phi(R,S) \subset S$, then $\mathsf{id}_D : R \subset \Delta$ and $\mathsf{id}_D : \Delta \subset S$. In particular, if Δ' is another invariant $\mathcal{R}$-relation for F then $\mathsf{id}_D : \Delta' \subset \Delta$ and $\mathsf{id}_D : \Delta \subset \Delta'$ implies $\Delta = \Delta'$, by normality.

2.5.4 Existence of Invariant $\mathcal{R}$-relations

A relational structure $\mathcal{R}$ is said to possess *inverse images* iff, for all $f : D \to E$ and $S \in \mathcal{R}(E)$, there is an $\mathcal{R}$-relation $f^*S \in \mathcal{R}(D)$ such that

$$g : R \subset f^*S \quad \Longleftrightarrow \quad fg : R \subset S$$

for all $g : C \to D$ and $R \in \mathcal{R}(C)$. $\mathcal{R}$ has *intersections* iff, for all $S \subseteq \mathcal{R}(E)$, there is an $\mathcal{R}$-relation $\cap S$ in $\mathcal{R}(E)$ satisfying

$$g : R \subset \cap S \quad \Longleftrightarrow \quad \forall S \in S.\ g : R \subset S$$

for all $g : D \to E$ and $R \in \mathcal{R}(D)$.

Suppose a relational structure $\mathcal{R}$ possesses inverse images and intersections. Then the admissible $\mathcal{R}$-relations are closed under both operations. For each D, $\mathcal{R}(D)$ is a complete lattice under the order induced by $\mathsf{id}_D : - \subset -$, with meets given by intersection. Note that f^*S and $\cap S$ are uniquely determined, and for each $f : D \to E$ in $\mathcal{C}$, the inverse image operator $f^* : \mathcal{R}(E) \to \mathcal{R}(D)$ is a monotone function. The operation $f \mapsto f^*$ satisfies $\mathsf{id}_D^* = \mathsf{id}_{\mathcal{R}(D)}$ and $(fg)^* = g^*f^*$.

Theorem 2.5.2 (Pitts 1996)**.** *Let $\mathcal{R}$ be a relational structure with inverse images and intersections on a bilimit-compact category $\mathcal{C}$. Let $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ be a locally continuous functor equipped with an admissible action Φ on $\mathcal{R}$, and let $i : F(D, D) \cong D$ be its minimal invariant. Then the invariant $\mathcal{R}$-relation for F exists.*

Sketch of Proof. Note that Δ is the invariant $\mathcal{R}$-relation for F iff $\Delta = \psi(\Delta, \Delta)$, where $\psi : \mathcal{R}_{adm}(D) \times \mathcal{R}_{adm}(D) \to \mathcal{R}_{adm}(D)$ is the map

$$\psi(R^-, R^+) \stackrel{def}{=} (i^{-1})^* \Phi(R^-, R^+)$$

In fact, by admissibility of the action of F on $\mathcal{R}$-relations, ψ determines a monotone endofunction $\psi^{\S}$ on the complete lattice $\mathcal{R}_{adm}(D)^{op} \times \mathcal{R}_{adm}(D)$, by

$$\psi^{\S}(R^-, R^+) \stackrel{def}{=} \langle \psi(R^+, R^-), \psi(R^-, R^+) \rangle$$

By the Tarski-Knaster fixed point theorem, $\psi^{\S}$ has a least (pre-)fixed point $\langle \Delta^-, \Delta^+ \rangle$. It remains to show $\Delta^- = \Delta^+$, i.e., $\mathsf{id}_D : \Delta^- \subset \Delta^+$ and $\mathsf{id}_D : \Delta^+ \subset \Delta^-$, for then $\Delta^+ = \psi(\Delta^+, \Delta^+)$ is the invariant $\mathcal{R}$-relation. The second inclusion $\mathsf{id}_D : \Delta^+ \subset \Delta^-$ follows from the least pre-fixed point property of $\langle \Delta^-, \Delta^+ \rangle$.

We show the first inclusion $\mathsf{id}_D : \Delta^- \subset \Delta^+$. By admissibility of Δ^+, the set

$$[\Delta^-, \Delta^+] = \{e \in \mathcal{C}(D, D) \mid e : \Delta^- \subset \Delta^+\}$$

is chain-closed and contains $\perp$. Moreover, by admissibility of the action of F and the definition of ψ, $[\Delta^-, \Delta^+]$ is closed under application of $\delta(e) = iF(e, e)i^{-1}$. By fixed point induction, $lfp(\delta) \in [\Delta^-, \Delta^+]$. Thus $\mathsf{id}_D : \Delta^- \subset \Delta^+$ since $lfp(\delta) = \mathsf{id}_D$ as D is the minimal invariant. □

Inspection of the proofs of Proposition 2.5.1 and Theorem 2.5.2 shows that the statement can be strengthened, by allowing the action of the functor to be defined on parts of the relational structure only (Reus and Streicher 2004). This result is used when defining the semantics of store specifications in Chapter 4, where a functional Φ is given only on relations $R \in \mathcal{R}(D)$ over the minimal invariant D. An even stronger requirement is used in Chapter 10, where admissibility of a functor action can be proved only with respect to projection maps.

Theorem 2.5.3. *Let $\mathcal{R}$ be a relational structure with inverse images and intersections on a bilimit-compact category $\mathcal{C}$. Let $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ be a locally continuous functor, and let $i : F(D,D) \cong D$ be its minimal invariant.*

Suppose $I \subseteq \mathcal{C}(D,D)$ is a chain-closed set that contains the projections $\delta^n(\bot)$. Moreover, suppose that $\Phi : \mathcal{R}_{adm}(D) \times \mathcal{R}_{adm}(D) \to \mathcal{R}_{adm}(D)$ is an operation such that for all $\mathcal{R}$-relations $R, S, R', S' \in \mathcal{R}_{adm}(D)$ and all $h \in I$,

$$h : R' \subset R \ \wedge \ h : S \subset S' \quad \Longrightarrow \quad F(h,h) : \Phi(R,S) \subset \Phi(R',S')$$

Then the invariant $\mathcal{R}$-relation for F exists and is uniquely determined.

Sketch of proof. In the statement of Proposition 2.5.1 let $A = B = D$. In the proof, replace the set $[R,\Delta] \times [\Delta,S]$ by the set

$$([R,\Delta] \times [\Delta,S]) \cap \mathit{diag}(I) \ = \ \{\langle h,h\rangle \mid h \in I \ \wedge \ h : R \subset \Delta \ \wedge \ h : \Delta \subset S\}$$

which is also a chain-closed subset of $\mathcal{C}(D,D)$ that contains $\langle \bot, \bot \rangle$. Uniqueness follows from this exactly as before.

Similarly, in the proof of Theorem 2.5.2 replace the set $[\Delta^-, \Delta^+]$ by the chain-closed set

$$[\Delta^-, \Delta^+] \cap I \ = \ \{e \in \mathcal{C}(D,D) \mid e \in I \ \wedge e : \Delta^- \subset \Delta^+\}$$

From the assumption of the theorem, $\delta^n(\bot) \in [\Delta^-, \Delta^+] \cap I$ for all n. Thus also $\mathsf{id}_D \in [\Delta^-, \Delta^+]$ which shows the inclusion $\Delta^- \subseteq \Delta^+$ as before. □

Chapter 3

A Model of Objects

In this chapter we formally introduce the object-oriented languages we are concerned with in subsequent chapters. We review some of the many encodings and translations of object-oriented concepts into more traditional, procedural constructs that have been proposed in the literature.

A variant of the imperative object calculus of Abadi and Cardelli (1996) is introduced, with simple and recursive type systems. Operational and denotational semantics are presented. A proof of adequacy follows the standard approach with respect to languages with recursive types, for instance, see (Pitts 1996). Abadi and Leino's (1997, 2004) logic, which we consider in Part II, talks about behaviour of programs of this calculus. Furthermore, Chapter 8 in Part III presents typed semantics for this language.

Finally, we provide syntax and semantics of a simple class-based language, based on the ideas and compilation techniques outlined in (Abadi and Cardelli 1996). This language is intended as a vehicle to investigate more advanced, dynamic aspects of class-based languages, which are outside the scope of most previous semantic models and logics of classes.

3.1 Class-based and Object-based Languages

The term "object-oriented" is used broadly. Two main directions are *class-based* and *object-based* programming languages; in practice, the former are used almost exclusively.

Objects, understood as entities with (hidden) internal state and methods operating on this state, are central to both approaches. Methods can refer to their "host" object, and thus to its fields and further methods, by a special variable, usually called `self` or `this`. In class-based languages, the concept of a *class* provides a template, describing the *interface* and *behaviour*, for the collection of its instance objects. Classes are extensible: *inheritance* allows us to extend the interface, and possibly adapt the behaviour of methods. In object-based languages, the class concept is replaced by constructs to directly construct individual objects from scratch, or *clone* and adapt existing objects.

While the latter may appear to be "simpler" by avoiding the separate concept of

classes, it has been remarked that this simplicity comes at a price (Reus 2002; Bruce 2002): In class-based languages, all method code is usually known in advance. Classes separate the method code from objects, the class *name* is used to select the appropriate code for method invocations from the *class table*. Semantically, this distinction between construction of the class table and program execution is often exploited by interpreting programs with respect to the fixed point of a given class table. In contrast, in object-based languages where objects can be created on-the-fly, every object contains its own suite of methods. Operationally speaking, the heap store for such a language contains code, i.e., is higher-order.

Thus, despite their *syntactic* simplicity, the *semantics* of object-based languages becomes much more involved in comparison to the class-based case. This is particularly true with regard to specification and verification of object-oriented programs: The design and soundness of most program logics that appeared in the literature rely on a "closed-world" assumption, i.e., that the class table defining the classes and their inheritance relationships is fixed. This is too restrictive for object-based languages which are inherently "open". Conversely, the closed-world assumption underlying many proposed semantic models of class-based languages means *compositionality* is problematic.

Reus (2002) argues that the difference between object-based and class-based languages manifests itself on the amount of domain-theoretic machinery necessary to reason about objects: While fixed-point induction is the main proof technique in the class-based case, specifications in the object-based case are generally defined by a mixed-variant recursion for which existence cannot be taken for granted (cf. Section 2.5 and the definition of store specifications in Chapter 4).

Advanced class-based features lead to similar semantic challenges, see the discussions in (Reus 2003; Reus 2002). We therefore consider object-based languages an ideal starting point for studying modularity issues that occur also in class-based languages. In fact, class-based programs can be compiled into object-based ones (Abadi and Cardelli 1996), and object-based languages can naturally deal with classes defined dynamically, like inner classes and classes loaded at run-time. See Section 3.4.

3.2 Object Encodings

A large amount of previous research investigated how object-oriented constructs could be explained in terms of – ideally well-understood – procedural features, by giving either a semantic model or a syntactic translation into typed lambda calculus. Both functional and imperative variants of objects have been looked at.

In the following we give an overview of the main variants of these encodings of objects, based on the surveys of Bruce, Cardelli, and Pierce (1999), Abadi and Cardelli (1996) and Kamin and Reddy (1994). In a sense, encodings of (imperative) objects into lambda calculi with *imperative* features can be much simpler than encodings in functional variants. This should not be surprising: The explicit modelling of the encapsulated state of objects, and the ability to faithfully simulate state changes, requires sophisticated type systems in functional target languages. This is the case even for relatively poor type systems on

the object side.

3.2.1 Imperative Objects

Apart from minor variations, one can distinguish two styles of encoding imperative objects, differing in the way the dependence of methods on the "self" parameter is resolved. Recall from the description in Section 3.1 that this parameter refers to the host object and allows one to access the fields and make calls to "sibling" methods from within a method body. The first possibility, called *self-application semantics* in (Kamin and Reddy 1994; Glimming and Ghani 2004), is to make this parameter explicit: Methods are encoded as procedures with an additional parameter s, i.e., a method declaration

$$\mathsf{m}(x_1, \ldots, x_n) = e \quad \text{translates to} \quad \mathsf{m} = \lambda s\, \lambda \langle x_1 \ldots x_n \rangle \,.\, e$$

Such procedures are often called *pre-methods*, and an object is simply a record of fields and pre-methods. At method invocation time, the self parameter s is bound to the host object o, so that

$$o.\mathsf{m}(v_1, \ldots, v_n) \quad \text{translates to} \quad o.\mathsf{m}(o)(v_1, \ldots, v_n)$$

The self-application semantics is close to operational intuition and can also easily model advanced non-standard object-oriented features such as method update (Abadi and Cardelli 1996).

The second possibility is to resolve the dependence on self immediately at object creation time, rather than at the time of method invocation: the record of pre-methods gives rise to objects where the methods are defined mutually recursively. More precisely, if $m = \{\!|\mathsf{m}_j = \lambda s \lambda x_j.e_j|\!\}$ is the record of pre-methods and r the (mutable) record of fields, then an object is obtained by applying the recursively defined constructor map $\mathit{create}(r)(m)$ where

$$\mathit{create}(r)(m) \;=\; \{\mathsf{f}_i = r.\mathsf{f}_i,\ \mathsf{m}_j = m.\mathsf{m}_j(\mathit{create}(r)(m))\}_{i,j} \tag{3.1}$$

In particular, the self parameter is not a formal parameter of methods in this interpretation. Kamin and Reddy call this encoding the *fixed-point*, or *closure*, model. The fixed point model of objects does not model object-oriented languages where method update is possible, since updates of methods are not visible to sibling methods.

Untyped vs. Typed Objects

Both encodings work well in the untyped case in that they provide an adequate model of objects. In the case of typed languages, however, the fixed-point model is preferable. In the self-application semantics, the translations of methods will be assigned types of the form $A \times B_1 \times \cdots \times B_n \Rightarrow B$ where A is the type of the object (the self parameter), $B_1, \ldots, B_n$ are the types of the formal parameters, and B is the result type of the method. The (contra-variant) occurrence of the object type A blocks desirable subtypings.

There exist, however, variations of the self-application semantics that fix this problem. In the *cyclic record* encoding (Eifrig, Smith, Trifonov, and Zwarico 1995), the technique of

"back-patching" (Landin 1964) is used. As in the fixed-point model, this means the self parameter is bound to the location of the object in the store at object creation time. So the type of methods is simply $B_1 \times \cdots \times B_n \Rightarrow B$, and does not contain a parameter of type A for the host object. In contrast to the fixed point model, and as in the self-application model, a call to a sibling method is resolved by executing the method found at the location of self in the store at method call time. In the presence of method update, this method could be different from the one available at object creation time. Abadi and Cardelli point out that in order to extend the cyclic record encoding to cloning constructs, methods would have to be abstracted on the location of self, bringing us back to the self-application model.

An encoding due to Abadi, Cardelli, and Viswanathan (1996), called *imperative self-application* semantics in (Abadi and Cardelli 1996), refines the self-application semantics in a different way, by using a more expressive type system with both recursive and bounded existential types. For instance, an object with a single field f of type A and a method m of type $B_1 \Rightarrow B_2$ is encoded as a record of type

$$\mu(Y)\, \exists(X \preceq Y)\, \{\mathsf{self} : X,\, \mathsf{f} : A,\, \mathsf{m} : X \times B_1 \Rightarrow B_2\} \tag{3.2}$$

The component self refers to the record itself and provides a value of the partially abstract type X. Method invocation is encoded by applying the component m to self. The use of the recursive type abstraction achieves the desired subtypings. This encoding also provides for cloning when a component clone of type $\{\} \Rightarrow X$ is added to the record. The procedure clone simply constructs another such record at a fresh location in memory.

3.2.2 Functional Objects

Even with a purely "functional" semantics, objects inherently embody a notion of private local state, through their fields. Usually this means an operation that performs updates on the state must return another object, of the same type as the self parameter, and so the notion of *self type* becomes a much more important concept compared to imperative object calculi. Consequently, the need to model state change caused by field update led to sophisticated encodings. The encodings differ considerably in the treatment of the self type.

As in the imperative case, one can classify the many encodings roughly into two cases, depending on how the self parameter of methods is treated. In the *recursive record* encoding (described as $OR(I)$ in (Bruce, Cardelli, and Pierce 1999)), objects are interpreted as recursively defined records. Methods do not take an explicit argument for self. In contrast, most other encodings rely on a variant of *self-application* encodings where methods require (sometimes only the state part of) the host object to be passed as an additional argument at method invocation.

The recursive record encoding works well for objects if only *internal* updates are allowed, i.e., all updates are of the form *this*.f:=*b* where *this* refers to the self parameter of the directly enclosing host object. Thus typical examples like movable (colour) points can be treated: The type of the encodings uses recursive record types, for instance the type

of point object encodings is *Point* where

$$Point \;=\; \{x : \mathsf{int},\ move : \mathsf{int} \Rightarrow Point\}$$

Color points have an additional field,

$$ColorPoint \;=\; \{c : \mathsf{color},\ x : \mathsf{int},\ move : \mathsf{int} \Rightarrow ColorPoint\}$$

The expected subtyping between points and colour points holds. However, a uniform treatment of arbitrary updates, such as those found in Abadi and Cardelli's object calculi, is lacking in this encoding.

An interesting refinement of the functional recursive record encoding is the *split method* encoding (Abadi, Cardelli, and Viswanathan 1996). The ability to access a component l (there is no need to distinguish between methods and fields in this encoding) of an object is split into distinct abilities l^{sel} and l^{upd}, for selecting and updating, respectively: An object type $[l_i : A_i \Rightarrow B_i]_i$ gives rise to a recursive record type A in the encoding, with

$$A \;=\; \{l_i^{sel} : A_i \Rightarrow B_i,\ l_i^{upd} : (A \Rightarrow A_i \Rightarrow B_i) \Rightarrow A\}_i$$

Note that all occurrences of A on the right hand side are covariant, so this encoding validates the expected subtyping rules for objects. On the level of terms, the encoding is based on a *create* function that constructs recursive records, similar to (3.1) above: The select components l_i^{sel} are thus bound to self at object construction time. The update components l_i^{upd} return updated objects, by applying the *create* function to the original record of pre-methods where the i-th pre-method has been replaced by the actual parameter of l_i^{upd}.

This line of work culminated in Viswanathan's (1998) article, providing an interpretation of functional objects that is *fully abstract*. There are interesting resemblances between the split method encoding and the work of Hofmann and Pierce (1996) on positive subtyping, where updating of functional records is axiomatised.

In the naive *self-application* encoding, methods are passed the host object as additional parameter at method invocation time, exactly as in the imperative self-application model. Consequently, an object type $\left[\mathsf{f}_i : A_i, \mathsf{m}_j : B_j \Rightarrow C_j\right]_{i,j}$ translates to the recursive type A with

$$A \;=\; \{\mathsf{f}_i : A_i,\ \mathsf{m}_j : A \times B_j \Rightarrow C_j\}_{i,j}$$

and, just as in the imperative self-application encoding, the contravariant occurrence of the type A in the type of methods blocks all subtypings. However, combining the self-application encoding with existential types provides a better solution. The state part of the object is "hidden", using the well-known connection between information hiding and abstract data types, and existential types (Mitchell and Plotkin 1988). An object type $\left[\mathsf{f}_i : A_i, \mathsf{m}_j : B_j \Rightarrow C_j\right]_{i,j}$ is encoded as

$$\exists X.\, X \times \{\mathsf{m}_j : X \Rightarrow B_j \Rightarrow C_j\}_j$$

where the state part is hidden by the existentially quantified X. In particular, the contravariant method arguments corresponding to the state part are abstracted too, so that this *state-application* encoding (explained as $OE(I)$ in (Bruce, Cardelli, and Pierce 1999)) validates desired subtypings between object types. Code that invokes a method must explicitly unpack, apply, and repackage the hidden state part. A variant of this encoding using also recursive types (referred to as $ORE(I)$ in *loc. cit.*), shifts this burden to the method bodies so that method calls are uniform.

A further refinement, similar to the typed imperative self-application encoding (3.2) above, uses both recursive types and bounded existential types. It is explained as $ORBE(I)$ encoding in (Bruce, Cardelli, and Pierce 1999), and used in Abadi and Cardelli's book (1996) to provide a foundational description of the notion of self type. The advantage over the somewhat simpler state-application encodings is that it also provides for method update, by implicitly passing not only the state part but the whole object at method invocations.

3.2.3 Models of Objects in this Thesis

In the remainder of this thesis, we are only concerned with semantics and logics of imperative object-oriented languages. The precise formalisation of objects is that of Abadi and Cardelli (1996) which is presented in Section 3.3 below.

The semantics of untyped objects we consider for the program logic in Part II is the self-application model. The reasons for this choice are twofold: Firstly, this corresponds closely to the operational model of Abadi and Cardelli (1996). Secondly, self-application can not be avoided because of the higher-order store: It is possible to write recursive methods where the recursion is "through the store", by calling methods of objects referenced by member fields (as in Example 1.3). The logic of Abadi and Leino can deal with such recursive functions. In view of this, we think it is reasonable to use recursion through the store also to implement the explicit recursion through self, and we avoid having to additionally treat explicit fixed points in the semantics.

Object types can be expressed as predicates over objects, as a special case of the specifications of Abadi and Leino's logic. The semantics of typed objects which our interpretation of the logic entails has some interesting similarities and differences with the imperative self-application variants mentioned above. This is discussed in more detail in Section 5.2.4. Note that in a previous paper (Reus and Schwinghammer 2005) we did not consider cloning constructs. The interpretation in (Reus and Schwinghammer 2005) essentially amounted to the cyclic record encoding.

Finally, the typed semantics of objects we obtain in Chapters 8 and 9 is based on the fixed-point model: This model uses fixed points to interpret recursion through self in order to avoid typing problems (cf. Section 3.2.1). Nevertheless, examples using recursion through the store can be written using assignment, so similar reasoning techniques are (sometimes) required.

Table 3.1 *Syntax of the imperative object calculus*

$a, b \in Ob$	::=	x	variable
	|	true | false	booleans
	|	if x then a else b	conditional
	|	let $x = a$ in b	let
	|	$[\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J}$	object construction
	|	$x.\mathsf{f}$	field selection
	|	$x.\mathsf{f} := y$	field update
	|	$x.\mathsf{m}$	method invocation
	|	clone x	shallow copy

3.3 Object Calculus

In order to develop a theoretical foundation for object-oriented programming languages, Abadi and Cardelli proposed a number of small calculi where objects are primitive (Abadi and Cardelli 1996). Later work showed that for many of these calculi good encodings into lambda calculi exist (Abadi, Cardelli, and Viswanathan 1996; Viswanathan 1998). However, Abadi and Cardelli (1996) argued that the complexity exhibited by these translations justifies the study of primitive objects, seeing that the abstractions provided by object-oriented languages are highly non-trivial.

Abadi and Cardelli's book considers objects with both functional and imperative semantics, and with type systems ranging from simple first-order types through recursive and second-order to higher-order types. To provide a streamlined, minimal model of objects, fields and methods are usually identified. As a consequence, not only fields but also methods may be updated.

In much of the technical parts of this thesis we adopt the imperative object calculus with first-order types as our notion of object-oriented language. More precisely, we consider the variant of the calculus as presented in (Abadi and Leino 2004) where fields and methods *are* distinguished; field update is possible but method update is not. In addition, the cloning construct of (Abadi and Cardelli 1996) is used. It does not significantly increase the complexity of the semantics, and is useful in transferring results from object-based to class-based languages. We also consider the extension with recursive first-order types, as a special case of the recursive specifications considered in Chapter 6.

In the remainder of this section the syntax and typing rules of this calculus are summarised. Both operational and denotational semantics are given.

3.3.1 Imperative Object Calculus

Syntax

Let Var, $\mathcal{M}$ and $\mathcal{F}$ be pairwise disjoint, countably infinite sets of *variables*, *method names* and *field names*, respectively. Let x, y, z range over Var, let $\mathsf{m} \in \mathcal{M}$ and $\mathsf{f} \in \mathcal{F}$. The language is defined by the grammar in Table 3.1.

Variables are (immutable) identifiers, the semantics of booleans and conditional is as usual. The object expression $\mathsf{let}\ x = a\ \mathsf{in}\ b$ first evaluates a and then evaluates b with identifier x bound to the result of a.

In constructing an object $\left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j\right]_{i\in I, j\in J}$ we assume all the labels f_i and m_j are distinct. Such an expression 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, the symbol ς is a binder, binding the explicit 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. If clear from the context we may omit the index sets I and J.

The result of field selection $x.\mathsf{f}$ is the value of the field, and $x.\mathsf{f} := y$ is a destructive update of this value. A formal semantics is given in the next subsections below.

The notions of free and bound variables are defined in the usual way; $\mathsf{fv}(a)$ denotes the set of free identifiers of a. We identify objects that differ only in the names of bound variables and the order of components. Finally, we write *Prog* for the set of *programs*, i.e., the subset of closed object terms:

$$\mathit{Prog} \stackrel{def}{=} \{a \in \mathit{Ob} \mid \mathsf{fv}(a) = \emptyset\}$$

Remark 3.3.1. *As mentioned above, in contrast to (Abadi and Cardelli 1996) we distinguish between fields and methods; method update is disallowed. Following (Abadi and Leino 2004) we also restrict the cases for field selection, field update, method invocation and conditional to contain only variables as subterms, instead of arbitrary object terms. This is no real limitation because of the* let *construct, but it simplifies the statement both of the semantics and the rules of the logic in Section 4.3: Firstly, control-flow is made explicit; secondly, except for the cases of* let *and* if-then-else*, evaluation of subterms is side-effect free. We use a more generous syntax (for instance, also allowing for natural numbers) in the examples.*

Example 3.3.2. We consider an object-based modelling of a bank account as an example:

$$\begin{aligned}
\mathit{acc}(x) \equiv [&\mathsf{balance} = 0,\\
&\mathsf{deposit10} = \varsigma(y)\ \mathsf{let}\ z = y.\mathsf{balance}{+}10\ \mathsf{in}\ y.\mathsf{balance}{:=}z,\\
&\mathsf{withdraw10} = \varsigma(y)\ \mathsf{let}\ z = y.\mathsf{balance}{-}10\ \mathsf{in}\ y.\mathsf{balance}{:=}z,\\
&\mathsf{interest} = \varsigma(y)\ \mathsf{let}\ r = x.\mathsf{manager.rate}\ \mathsf{in}\\
&\qquad\qquad \mathsf{let}\ z = y.\mathsf{balance}{\times}r/100\ \mathsf{in}\ y.\mathsf{balance}{:=}z]
\end{aligned}$$

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. □

We will come back to this example in Section 4 where we give a specification of such bank account objects in terms of Abadi and Leino's logic.

First-Order Types

We briefly recall the system of first-order types for the imperative object calculus (Abadi and Cardelli 1996, Chap. 11). The syntax of types object types is given by the grammar

$$A, B \in \mathit{Type} ::= \mathsf{bool} \mid \left[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j\right]_{i\in I, j\in J}$$

Table 3.2 *Typing rules for the imperative object calculus*

(Term Sub) $$\frac{\Gamma \rhd a : B \qquad B \preceq A}{\Gamma \rhd a : A}$$

(Term Var) $$\frac{}{\Gamma, x{:}A, \Gamma' \rhd x : A}$$

(Term Const) $$\frac{}{\Gamma \rhd \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \rhd \mathsf{false} : \mathsf{bool}}$$

(Term Cond) $$\frac{\Gamma \rhd x : \mathsf{bool} \qquad \Gamma \rhd a : A \qquad \Gamma \rhd b : A}{\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b : A}$$

(Term Let) $$\frac{\Gamma \rhd a : A \qquad \Gamma, x{:}A \rhd b : B}{\Gamma \rhd \mathsf{let}\ x = a\ \mathsf{in}\ b : B}$$

(Term Obj) $$\frac{A \equiv [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \qquad \Gamma \rhd x_i : A_i \quad \forall i \in I \qquad \Gamma, y_j{:}A \rhd b_j : B_j}{\Gamma \rhd [\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j) b_j]_{i\in I, j\in J} : A}$$

(Term Sel) $$\frac{\Gamma \rhd x : [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \qquad k \in I}{\Gamma \rhd x.\mathsf{f}_k : A_k}$$

(Term Upd) $$\frac{\Gamma \rhd x : [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \qquad k \in I \qquad \Gamma \rhd y : A_k}{\Gamma \rhd x.\mathsf{f}_k := y : \mathbf{1}}$$

(Term Inv) $$\frac{\Gamma \rhd x : [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \qquad k \in J}{\Gamma \rhd x.\mathsf{m}_k : B_k}$$

(Term Copy) $$\frac{A \equiv [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \qquad \Gamma \rhd x : A}{\Gamma \rhd \mathsf{clone}\ x : A}$$

bool is the type of truth values. An object has type $\left[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j\right]_{i\in I, j\in J}$ if it has fields f_i of type A_i and methods m_j returning results of type B_j, for all $i \in I$ and $j \in J$. As for values, we identify object types that differ only in the order of their components.

The subtype relation $A \preceq B$ is the least reflexive, transitive relation closed under the rule

(Sub Obj) $$\frac{B_j \preceq C_j \quad \forall j \in J' \qquad I' \subseteq I \qquad J' \subseteq J}{[\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \preceq [\mathsf{f}_i : A_i, \mathsf{m}_j : C_j]_{i\in I', j\in J'}}$$

Thus, subtyping on objects is by width, and for methods also by depth. It is well-known that the invariance in the types of fields is essential for type soundness.

As usual, typing judgements are of the form $\Gamma \rhd a : A$ where Γ is a *context*, i.e., a finite functional relation between *Var* and *Type*. The typing rules are given in Table 3.2 where we set $\mathbf{1} \stackrel{def}{=} [\,]$. By (Sub Obj) and the subsumption rule, (Term Sub), every typable object has this type. The rules for variables, constants, conditional and let are standard as in typed lambda calculi (Pierce 2002). In the case (Term Obj) the body b_j of a method m_j is typed under the assumption that the self parameter y_j has type A, the type of the object being constructed. The rules (Term Sel) and (Term Inv) of field selection and method invocation, resp., amount to checking that a component, named with the selected label and of appropriate type, exists. The case of field update, (Term Upd), additionally guarantees that the new value has a type equal to the original one. Finally, cloning applies to any object type A.

Table 3.3 *Well-formed recursive object types*

(CONTEXT EMP)	$\dfrac{}{\emptyset \rhd \mathsf{ok}}$
(CONTEXT VAR)	$\dfrac{\Delta \rhd Y \quad X \notin \mathsf{dom}(\Delta)}{\Delta, X \preceq Y \rhd \mathsf{ok}}$
(CONTEXT TOP)	$\dfrac{\Delta \rhd \mathsf{ok} \quad X \notin \mathsf{dom}(\Delta)}{\Delta, X \preceq \top \rhd \mathsf{ok}}$
(TYPE VAR)	$\dfrac{\Delta, X \preceq A, \Delta' \rhd \mathsf{ok}}{\Delta, X \preceq A, \Delta' \rhd X}$
(TYPE CONST)	$\dfrac{\Delta \rhd \mathsf{ok}}{\Delta \rhd \top} \qquad \dfrac{\Delta \rhd \mathsf{ok}}{\Delta \rhd \mathsf{bool}}$
(TYPE OBJ)	$\dfrac{\Delta \rhd A_i \quad \forall i \in I \qquad \Delta \rhd B_j \quad \forall j \in J}{\Delta \rhd [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j]_{i\in I, j\in J}}$
(TYPE REC)	$\dfrac{\Delta, X \preceq \top \rhd A}{\Delta \rhd \mu(X)A}$

Recursive Types

Recursive types are necessary when a field of an object or a result of one of the object's methods are supposed to have the same type as the object itself. In particular, they are needed to implement recursive data types such as lists and trees in the object calculus. We now extend the set *Type* of types by recursive types $\mu(X)A$, describing a type solving the equation $X = A$ where X may occur in A.

To prevent meaningless types such as $\mu(X)X$, type recursion is only allowed through object types, thereby enforcing "formal contractiveness":

$$\begin{aligned} \underline{A}, \underline{B} \quad &::= \top \mid \mathsf{bool} \mid [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \mid \mu(X)\underline{A} \\ A, B \in \mathit{RecType} &::= \underline{A} \mid X \end{aligned}$$

where X ranges over an infinite set *TyVar* of type variables. The reason for introducing $\top$ is that for the type of empty objects $\mathbf{1}$ the subtyping $\mathsf{bool}{<:}\mathbf{1}$ does *not* hold: Below we will turn the type $\top$ into the greatest type in the subtype ordering. X is bound in $\mu(X)A$, and as usual we identify types up to the names of bound variables.

In addition to contexts Γ of term variables, we introduce contexts Δ of type variables with an upper bound, $X \preceq A$, where A is either another variable or $\top$. We may simply write Δ, X, Δ' for $\Delta, X \preceq \top, \Delta'$ if type variables have trivial upper bounds.

In the type inference rules of Table 3.2 every judgement $\Gamma \rhd a : A$ is replaced by the judgement $\Gamma; \Delta \rhd a : A$. We name the rules in this system accordingly: For instance, we write (RECTERM VAR) in place of (TERM VAR), similarly for the other rules. Moreover, there are now the additional judgements $\Delta \rhd A$ and $\Delta \rhd \mathsf{ok}$ defining well-formed types and well-formed type contexts, respectively. These are given in Table 3.3.

The subtype relation is now defined with respect to a type environment Δ, see Table 3.5; Table 3.4 contains the subsumption rule extended with assumptions Δ. Subtyping of recursive types is obtained by the rule (RECSUB REC) considered in (Amadio and

Table 3.4 *Subsumption for recursive object types*

(RecTerm Sub)	$\dfrac{\Gamma;\Delta \rhd a : B \qquad \Delta \rhd B \preceq A}{\Gamma;\Delta \rhd a : A}$

Table 3.5 *Subtyping recursive object types*

(RecSub Top)	$\dfrac{\Delta \rhd A}{\Delta \rhd A \preceq \top}$
(RecSub Obj)	$\dfrac{\Delta \rhd B_j \preceq C_j \quad \forall\, j \in J' \qquad I' \subseteq I \qquad J' \subseteq J}{\Delta \rhd [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j]_{i\in I, j\in J} \preceq [\mathsf{f}_i : A_i, \mathsf{m}_j : C_j]_{i\in I', j\in J'}}$
(RecSub Rec)	$\dfrac{\Delta, Y \preceq \top, X \preceq Y \rhd A \preceq B}{\Delta \rhd \mu X.A \preceq \mu Y.B}$
(RecSub Var)	$\dfrac{\Delta, X \preceq A, \Delta' \rhd \mathsf{ok}}{\Delta, X \preceq A, \Delta' \rhd X \preceq A}$
(RecSub Fold)	$\dfrac{\Delta \rhd \mu X.A}{\Delta \rhd A[(\mu X.A)/X] \preceq \mu X.A}$
(RecSub Unfold)	$\dfrac{\Delta \rhd \mu X.A}{\Delta \rhd \mu X.A \preceq A[(\mu X.A)/X]}$

Cardelli 1993). The inference (RecSub Top) makes $\top$ the maximal element in the subtype preorder. The subtype relation is also used to fold and unfold recursive types, using the rules (RecSub Fold) and (RecSub Unfold).

Remark 3.3.3. *Our treatment of recursive object types is similar to the system for functional objects in Chapter 9 of (Abadi and Cardelli 1996). However, we depart from loc. cit. by not allowing arbitrary types as upper bounds in type contexts Δ. That is, we consider recursive rather than bounded recursive types. Moreover, we did not introduce* fold *and* unfold *terms to mediate between a recursive type $\mu(X)A$ and its unfolding $A[\mu(X)A/X]$. Semantically this amounts to solving the type equation $X = A$ up to equality rather than isomorphism: In the semantics of Chapter 6 where types (and specifications) denote predicates over an untyped value space, $\mu(X)A$ and $A[\mu(X)A/X]$ denote the same predicate.*

3.3.2 Operational Semantics

The operational semantics renders more precise the informal description of object terms given above. Although a denotational semantics is used in later chapters, we think presenting the operational semantics as well is useful for two reasons: Firstly, the adequacy proof (Section 3.3.4) illustrates a concrete application of the abstract framework of Chapter 2 to a well-understood problem; the relational structures used in subsequent chapters tend to be more complicated. Secondly, once adequacy has been established, our soundness proof of Abadi and Leino's logic (Chapter 5) can be linked to the original presentation (Abadi and Leino 2004) that was based on an operational semantics.

The semantics relates terms a to *values* v, meaning that, operationally, executing a yields v. We need the following ingredients:

- Locations l denoting locations in the heap, drawn from a countably infinite set *Loc*.
- Runtime values, which can be stored, bound to identifiers, and returned as the results of computations. They are the booleans and locations, and ranged over by v, i.e., $v \in \mathit{Val} ::= \mathsf{true} \mid \mathsf{false} \mid l$
- Stacks $\boldsymbol{\rho}$, associating identifiers to values.
- Method closures c are pairs of methods and stacks, $\langle \varsigma(y)b, \boldsymbol{\rho} \rangle$, where the stack $\boldsymbol{\rho}$ provides value bindings for the free identifiers of $\varsigma(y)b$.
- Object closures $o \equiv [\mathsf{f}_i = v_i, \mathsf{m}_j = c_j]_{i,j}$, providing (runtime) values for the fields and method closures for the methods of the object.
- Heaps $\boldsymbol{\sigma}$, associating (a finite number of) locations to object closures.

Stacks and heaps are written as finite sequences $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ and $l_1 \mapsto o_1, \ldots, l_m \mapsto o_m$ where all x_i and all l_j are distinct. We write *Clos* for the set of method closures, *Env* for the set of stacks and *Store* for the set of heaps, and use the notation

$$\boldsymbol{\rho}, x \mapsto v \quad \text{and} \quad \boldsymbol{\sigma}, l \mapsto o \tag{3.3}$$

to describe extension of the stack (the heap, resp.) by a new binding that maps x to v (l to o, respectively). In (3.3) we make the implicit assumption that there is no binding for x in $\boldsymbol{\rho}$, nor for l in $\boldsymbol{\sigma}$. We write $\boldsymbol{\rho}(x)$ for the value v where $\boldsymbol{\rho}$ is of the form $\boldsymbol{\rho}', x \mapsto v, \boldsymbol{\rho}''$; note that v is uniquely determined if a binding for x exists. Similar notation is used for heaps, writing $\boldsymbol{\sigma}.l$ for the object closure stored at l, and for $\boldsymbol{\sigma} = l_1 \mapsto o_1, \ldots, l_n \mapsto o_n$ we write $\mathsf{dom}(\boldsymbol{\sigma})$ for the set of locations $\{l_1, \ldots, l_n\}$. Finally, if $l \in \mathsf{dom}(\boldsymbol{\sigma})$ then $\boldsymbol{\sigma}.l \leftarrowtail o$ denotes the heap that is obtained from $\boldsymbol{\sigma}$ by replacing the binding for location l by o.

Heaps and stacks serve different purposes: while the stack implements the *static* lexical scoping of let-bound identifiers and (possibly nested) self-parameters, the heap grows *dynamically* whenever a new object is created during a computation.

This store model differs from the one presented by Abadi and Cardelli (1996) and used in (Abadi and Leino 2004): In their model, a heap location contains individual *method closures*, i.e., pairs of methods and stacks. Accordingly, in the definition of values in (Abadi and Cardelli 1996), locations are replaced by records of locations. Instead, our store model corresponds to that of Reus and Streicher's (2004) denotational semantics, and more in accordance with implementations of object-oriented languages.

The operational semantics is a relation between a stack, an initial store and term a, and a result consisting of a store and a value, written

$$\boldsymbol{\rho}; \boldsymbol{\sigma} \rhd a \longmapsto v; \boldsymbol{\sigma}'$$

It is defined by the system of inference rules in Table 3.6 which, modulo the differing store model, agrees with the one of (Abadi and Cardelli 1996).

Table 3.6 *Operational semantics of imperative objects*

(Red Var) $$\frac{\boldsymbol{\rho}(x) = v}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd x \longmapsto v;\boldsymbol{\sigma}}$$

(Red Val) $$\frac{}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd v \longmapsto v;\boldsymbol{\sigma}}$$

(Red Cond1) $$\frac{\boldsymbol{\rho}(x) = \mathsf{true} \qquad \boldsymbol{\rho};\boldsymbol{\sigma} \rhd a \longmapsto v;\boldsymbol{\sigma}'}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b \longmapsto v;\boldsymbol{\sigma}'}$$

(Red Cond2) $$\frac{\boldsymbol{\rho}(x) = \mathsf{false} \qquad \boldsymbol{\rho};\boldsymbol{\sigma} \rhd b \longmapsto v;\boldsymbol{\sigma}'}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b \longmapsto v;\boldsymbol{\sigma}'}$$

(Red Let) $$\frac{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd a \longmapsto v';\boldsymbol{\sigma}'' \qquad \boldsymbol{\rho}, x{\mapsto}v';\boldsymbol{\sigma}'' \rhd b \longmapsto v;\boldsymbol{\sigma}'}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd \mathsf{let}\ x = a\ \mathsf{in}\ b \longmapsto v;\boldsymbol{\sigma}'}$$

(Red Obj) $$\frac{l \notin \mathsf{dom}(\boldsymbol{\sigma}) \qquad \boldsymbol{\rho}(x_i) = v_i \quad \forall i \in I \qquad \boldsymbol{\sigma}' = \boldsymbol{\sigma}, l \mapsto [\mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho} \rangle]_{i\in I, j\in J}}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd [\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j) b_j]_{i\in I, j\in J} \longmapsto l;\boldsymbol{\sigma}'}$$

(Red Sel) $$\frac{\boldsymbol{\rho}(x) = l \qquad \boldsymbol{\sigma}(l) = \left[\mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho}_j \rangle\right]_{i\in I, j\in J} \qquad k \in I}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd x.\mathsf{f}_k \longmapsto v_k;\boldsymbol{\sigma}}$$

(Red Upd) $$\frac{\boldsymbol{\rho}(x) = l \qquad \boldsymbol{\sigma}(l) = \left[\mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho}_j \rangle\right]_{i\in I, j\in J} \qquad k \in I \qquad \boldsymbol{\rho}(y) = v \qquad \boldsymbol{\sigma}' = \boldsymbol{\sigma}.l \leftarrow \left[\mathsf{f}_k = v, \mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho}_j \rangle\right]_{k\neq i\in I, j\in J}}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd x.\mathsf{f}_k := y \longmapsto l;\boldsymbol{\sigma}'}$$

(Red Inv) $$\frac{\boldsymbol{\rho}(x) = l \qquad \boldsymbol{\sigma}(l) = \left[\mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho}_j \rangle\right]_{i\in I, j\in J} \qquad k \in J \qquad (\boldsymbol{\rho}_k, y_k \mapsto l);\boldsymbol{\sigma} \rhd b_k \longmapsto v;\boldsymbol{\sigma}'}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd x.\mathsf{m}_k \longmapsto v;\boldsymbol{\sigma}'}$$

(Red Copy) $$\frac{\boldsymbol{\rho}(x) = l' \qquad \boldsymbol{\sigma}(l') = o \qquad l \notin \mathsf{dom}(\boldsymbol{\sigma}) \qquad \boldsymbol{\sigma}' = \boldsymbol{\sigma}, l \mapsto o}{\boldsymbol{\rho};\boldsymbol{\sigma} \rhd \mathsf{clone}\ x \longmapsto l;\boldsymbol{\sigma}'}$$

Abadi and Cardelli (1996) prove a subject reduction theorem to establish type soundness of the language with respect to the first-order types of Section 3.3.1,

$$\emptyset \rhd a : A \ \text{ and } \ \boldsymbol{\rho};\boldsymbol{\sigma} \rhd a \longmapsto v;\boldsymbol{\sigma}' \quad \Longrightarrow \quad \emptyset \rhd v : A \tag{3.4}$$

The actual statement, and the proof thereof, require the generalisation to open terms and the additional concept of *store typings*. Also, because of the use of locations as runtime values, the typing rules of Tables 3.2 and 3.4 have to be extended and interpreted with respect to such store typings.

Rather than repeating these definitions and the proof of (3.4) for recursive types and our slightly different store model, we defer all considerations of type soundness until the next part of the thesis. In fact, type soundness will be a corollary to the soundness proof of the *logic* of objects of Abadi and Leino (2004). While our soundness proof is with respect to a denotational semantics, store typings and their generalisation to *store specifications* will again play an important rôle.

3.3.3 Denotational Semantics of the Imperative Object Calculus

We give a denotational semantics of the imperative object calculus, working in the category **pCpo** of cpos and partial continuous functions. The language of the previous section finds its interpretation within the following system of recursively defined cpos:

$$\begin{aligned} \mathsf{Val} &= \mathsf{BVal} + \mathsf{Loc} \\ \mathsf{St} &= \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Ob}) \\ \mathsf{Ob} &= \mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Cl}) \\ \mathsf{Cl} &= \mathsf{Loc} \times \mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St} \end{aligned} \tag{3.5}$$

Loc is the countably infinite set of locations *Loc* viewed as discrete cpo, and BVal is the set of truth values *true* and *false*, considered as discrete cpo. Note how the domains in (3.5) reflect the operational semantics of Section 3.3.2:

- Values are either booleans or locations. In particular, Val is a discretely ordered countable cpo.
- Stores $\sigma \in \mathsf{St}$ are the semantic analogue of heaps $\boldsymbol{\sigma}$. A store consists of finitely many "allocated" locations, each associated with the object stored at this location.
- Objects $\langle r_F, r_M \rangle \in \mathsf{Ob}$ are the semantic counterpart of object closures $\langle a, \rho \rangle$. They consist of separate records for the fields and methods, respectively. This distinction between field and method parts allows us to easily identify the values and the ("non-flat") method code of an object; see below.
- As in the operational semantics, each method $h \in \mathsf{Cl}$ of an object depends on the location of the host object, used to interpret the self-parameter, and an initial store. In case of termination a method call yields a result value and a new store.

We also introduce a case for exceptional termination, error. In the denotational semantics of the object calculus this is used to model runtime errors, such as a method-not-understood due to a method invocation $x.\mathsf{m}$ on an object that does not provide a method with label m. Type soundness amounts to showing that the denotation of any well-typed program is different from error.

In order to find cpos satisfying (3.5) consider the following functor $F_{Store} : \mathbf{pCpo}^{op} \times \mathbf{pCpo} \to \mathbf{pCpo}$. It is obtained from (3.5) by solving for St and then separating positive and negative occurrences of St in the right-hand side,

$$F_{Store}(S,T) \stackrel{def}{=} \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}) \times \mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times S \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times T)) \tag{3.6}$$

It follows that F_{Store} is a locally continuous bifunctor. Recall from Chapter 2 that **pCpo** is bilimit-compact. Thus, by Theorem 2.3.2 there exists a minimal invariant solution St (uniquely determined up to unique isomorphism) such that $F_{Store}(\mathsf{St},\mathsf{St}) \cong \mathsf{St}$. For convenience we omit the isomorphisms and consider $F_{Store}(\mathsf{St},\mathsf{St}) = \mathsf{St}$ as equality in the following.

Let $\mathsf{Env} \stackrel{def}{=} \mathit{Var} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$ be the set of *environments*, i.e., maps between *Var* and Val with finite domain. We use a similar notation to the one for records in Chapter 2 to denote the update and extension of environments: If $\rho \in \mathsf{Env}$ and $v \in \mathsf{Val}$ then $\rho[x := v]$ is the

Table 3.7 *Denotational semantics of imperative objects*

$$[\![x]\!]\,\rho\sigma \stackrel{def}{=} \begin{cases} \langle \rho(x), \sigma\rangle & \text{if } x \in \mathsf{dom}(\rho) \\ \langle \mathsf{error}, \sigma\rangle & \text{otherwise} \end{cases}$$

$$[\![\mathsf{true}]\!]\,\rho\sigma \stackrel{def}{=} \langle \mathit{true}, \sigma\rangle$$

$$[\![\mathsf{false}]\!]\,\rho\sigma \stackrel{def}{=} \langle \mathit{false}, \sigma\rangle$$

$$[\![\mathsf{if}\ x\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2]\!]\,\rho\sigma \stackrel{def}{=} \begin{cases} [\![b_1]\!]\,\rho\sigma' & \text{if } [\![x]\!]\,\rho\sigma = \langle \mathit{true}, \sigma'\rangle \\ [\![b_2]\!]\,\rho\sigma' & \text{if } [\![x]\!]\,\rho\sigma = \langle \mathit{false}, \sigma'\rangle \\ \langle \mathsf{error}, \sigma'\rangle & \text{if } [\![x]\!]\,\rho\sigma = \langle v, \sigma'\rangle \text{ for } v \notin \mathsf{BVal} \end{cases}$$

$$[\![\mathsf{let}\ x = a\ \mathsf{in}\ b]\!]\,\rho\sigma \stackrel{def}{=} \mathbf{let}\ \langle v, \sigma'\rangle = [\![a]\!]\,\rho\sigma\ \mathbf{in}\ [\![b]\!]\,\rho[x := v]\sigma'$$

$$\left[\!\!\left[[\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J}\right]\!\!\right]\rho\sigma$$

$$\stackrel{def}{=} \begin{cases} \langle l, \sigma[l := \langle o_1, o_2\rangle]\rangle & \text{if } x_i \in \mathsf{dom}(\rho) \quad \forall i \in I \\ \langle \mathsf{error}, \sigma\rangle & \text{otherwise} \end{cases}$$

$$\text{where} \quad \begin{cases} l \notin \mathsf{dom}(\sigma) \\ o_1 = \{\!|\mathsf{f}_i = \rho(x_i)|\!\}_{i\in I} \\ o_2 = \{\!|\mathsf{m}_j = \lambda\langle l, \sigma\rangle.\,[\![b_j]\!]\,\rho[y_j := l]\sigma|\!\}_{j\in J} \end{cases}$$

$$[\![x.\mathsf{f}]\!]\,\rho\sigma \stackrel{def}{=} \mathbf{let}\ \langle l, \sigma'\rangle = [\![x]\!]\,\rho\sigma\ \mathbf{in} \begin{cases} \langle \sigma'.l.\mathsf{f}, \sigma'\rangle & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{f} \in \mathsf{dom}(\sigma'.l) \\ \langle \mathsf{error}, \sigma'\rangle & \text{otherwise} \end{cases}$$

$$[\![x.\mathsf{f} := y]\!]\,\rho\sigma \stackrel{def}{=} \mathbf{let}\ \langle l, \sigma'\rangle = [\![x]\!]\,\rho\sigma\ \mathbf{in\ let}\ \langle v, \sigma''\rangle = [\![y]\!]\,\rho\sigma'\ \mathbf{in} \begin{cases} \langle l, \sigma''[l{:=}\sigma''.l[\mathsf{f}{:=}v]]\rangle & \text{if } l \in \mathsf{dom}(\sigma'') \text{ and } \mathsf{f} \in \mathsf{dom}(\sigma''.l) \\ \langle \mathsf{error}, \sigma''\rangle & \text{otherwise} \end{cases}$$

$$[\![x.\mathsf{m}]\!]\,\rho\sigma \stackrel{def}{=} \mathbf{let}\ \langle l, \sigma'\rangle = [\![x]\!]\,\rho\sigma\ \mathbf{in} \begin{cases} \sigma'.l.\mathsf{m}(l, \sigma') & \text{if } l \in \mathsf{dom}(\sigma') \text{ and } \mathsf{m} \in \mathsf{dom}(\sigma'.l) \\ \langle \mathsf{error}, \sigma'\rangle & \text{otherwise} \end{cases}$$

$$[\![\mathsf{clone}\,x]\!]\,\rho\sigma \stackrel{def}{=} \mathbf{let}\ \langle v, \sigma'\rangle = [\![x]\!]\,\rho\sigma\ \mathbf{in} \begin{cases} \langle l, \sigma'[l := \sigma'.v]\rangle & \text{if } v \in \mathsf{Loc} \cap \mathsf{dom}(\sigma') \\ \langle \mathsf{error}, \sigma'\rangle & \text{otherwise} \end{cases}$$

$$\text{where} \quad l \notin \mathsf{dom}(\sigma')$$

environment that maps x to v, and for all $y \neq x$ is equal to $\rho(y)$ if this is defined, and undefined otherwise.

Given an environment $\rho \in \mathsf{Env}$, Table 3.7 gives the semantic equations for the interpretation $[\![a]\!]\,\rho$ of an object expression a,

$$[\![a]\!]\,\rho : \mathsf{St} \to (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St}$$

In the semantic equations a (semantic) strict **let** is used that is also "strict" with respect to error:

$$\mathbf{let}\ \langle v, \sigma\rangle = s\ \mathbf{in}\ s' \stackrel{def}{=} \begin{cases} \text{undefined} & \text{if } s \text{ is undefined} \\ \langle \mathsf{error}, \sigma'\rangle & \text{if } s = \langle \mathsf{error}, \sigma'\rangle \\ (\lambda\langle v, \sigma\rangle.s')\,s & \text{otherwise} \end{cases}$$

In addition, the following notational conventions are used throughout this thesis: 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 $\mathsf{Val}+\{\mathsf{error}\}$, writing simply l instead of $\mathsf{in}_{\mathsf{Loc}}(l)$ etc.

Remark 3.3.4. *Observe that, in contrast to (Reus and Streicher 2004), we distinguish between non-termination (undefinedness) and exceptional termination,* error*. Also note that because* Loc *is assumed to be infinite, the condition $l \notin \mathit{dom}(\sigma)$ in the case for object creation can always be satisfied. Therefore object creation will never raise* error *due to this negative condition. The same holds for object cloning.*

Remark 3.3.5. *Other variants of cloning are conceivable. In particular, from a (space) efficiency point of view, it may be more realistic to copy fields, but only forward method calls to the method of the prototype object:*

$$[\![\mathsf{clone}\, x]\!]\,\rho\sigma \stackrel{def}{=} \textbf{let } \langle v, \sigma'\rangle = [\![x]\!]\,\rho\sigma \textbf{ in } \begin{cases} \langle l, \sigma'[l := o]\rangle & \text{if } v \in \mathsf{Loc} \cap \mathit{dom}(\sigma') \\ \langle \mathsf{error}, \sigma'\rangle & \text{otherwise} \end{cases}$$

where $o \stackrel{def}{=} \{\!|\mathsf{f}_i = v.\mathsf{f}_i,\ \mathsf{m}_j = \lambda\langle l, \sigma\rangle.\,\sigma.v.\mathsf{m}_j(l,\sigma)|\!\}_{i,j}$ and $l \notin \mathit{dom}(\sigma')$. The difference between these variants would become observable if method update was possible.

Projecting Stores

We will make use of a projection to the part of the store that contains just data in Val, thus "forgetting" all closures of objects residing in the store. This projection $\pi_{\mathrm{Val}} : \mathsf{St} \to \mathsf{St}_{\mathsf{Val}}$ is defined by

$$(\pi_{\mathrm{Val}}\,\sigma).l.\mathsf{f} \stackrel{def}{=} \sigma.l.\mathsf{f}$$

for all $l \in \mathsf{Loc}$ and $\mathsf{f} \in \mathcal{F}$, where $\mathsf{St}_{\mathsf{Val}} \stackrel{def}{=} \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val}))$. We refer to $\pi_{\mathsf{Val}}(\sigma)$ as the *flat part* of σ. Note that for all $\sigma, \sigma' \in \mathsf{St}$,

$$\sigma \sqsubseteq_{\mathsf{St}} \sigma' \quad \Longrightarrow \quad \pi_{\mathrm{Val}}(\sigma) = \pi_{\mathrm{Val}}(\sigma') \tag{3.7}$$

Since Val is discretely ordered, so are $\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val})$ and $\mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{F}}(\mathsf{Val})) = \mathsf{St}_{\mathsf{Val}}$ from which implication (3.7) immediately follows.

3.3.4 Adequacy

Computational soundness and *adequacy* relate different semantics of a language (e.g., an operational and denotational one), by establishing agreement with respect to termination of programs (in an operational semantics) and definedness (in a denotational model).

A number of adequacy results have been obtained previously for object calculi:

- Aceto et al. (2000) prove adequacy of a functional variant of the object calculus with recursive first-order types, with respect to the per model presented in Abadi and Cardelli's book (1996).
- Glimming in his thesis (2005) proves adequacy for a functional object calculus with self-types, but without subtyping, with respect to an operationally defined translation into *FPC* (Fiore 1996). Thus, composing this translation with an interpretation in adequate models of *FPC* yields adequate models of his calculus.

Table 3.8 *Contexts*

$C \in Ctxt$	::=	$\bullet \mid \mathsf{if}\ x\ \mathsf{then}\ C\ \mathsf{else}\ b \mid \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ C \mid \mathsf{let}\ x = C\ \mathsf{in}\ b$
	$\mid$	$\mathsf{let}\ x = a\ \mathsf{in}\ C \mid [\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j, \mathsf{m} = \varsigma(x)C]_{i\in I, j\in J}$

- In his M.Sc. thesis, Fecher (1999) considers adequacy of both functional and imperative untyped object calculi with regards to **pCpo**-models. While these are similar to the model of Section 3.3.3 above, they correspond to the different store model of Abadi and Cardelli.

In this section, we prove adequacy of the denotational model with respect to the operational semantics of Section 3.3.2. To the best of our knowledge, no adequacy proof for our semantics appears in the literature (although it is very similar to (Fecher 1999)). The details of the proof are complicated due to the recursively defined domain St, but follow the standard approach for languages with recursive types by exhibiting a *formal approximation relation* (see for example Pitts 1996). This relation is defined by a mixed-variant recursion, thus calling for the results of Chapter 2.

Contextual Equivalence

It is easy to verify that evaluation of programs (i.e., closed object terms) does not depend on the stack. Thus we may omit the environment in both the operational and denotational semantics. For instance, we simply write $\boldsymbol{\sigma} \triangleright a \longmapsto v; \boldsymbol{\sigma}'$ in place of $\boldsymbol{\rho}; \boldsymbol{\sigma} \triangleright a \longmapsto v; \boldsymbol{\sigma}'$ and $[\![a]\!]$ instead of $[\![a]\!]\,\rho$. For programs a we define a termination relation,

$$\boldsymbol{\sigma} \triangleright a \Downarrow \quad \stackrel{def}{\Longleftrightarrow} \quad \exists v \in Val\, \exists \boldsymbol{\sigma}' \in Store.\quad \boldsymbol{\sigma} \triangleright a \longmapsto v; \boldsymbol{\sigma}'$$

Table 3.8 defines *(single-hole) contexts* as extended object terms with a "hole", where a single subterm has been replaced by an occurrence of the distinguished, otherwise unused, identifier $\bullet$. Substitution of a term a for $\bullet$ in C is written $C[a]$; this substitution may capture free variables of a.

Definition 3.3.6 (Contextual Approximation)**.** *Let $a, b \in Ob$. A* contextual approximation *relation $\lesssim \subseteq Ob \times Ob$ is defined by*

$$a \lesssim b \quad \stackrel{def}{\Longleftrightarrow} \quad \forall C \in Ctxt\ \forall \boldsymbol{\sigma} \in Store.\ C[a], C[b] \in Prog \implies (\boldsymbol{\sigma}; C[a] \Downarrow \implies \boldsymbol{\sigma}; C[b] \Downarrow)$$

Contextual equivalence, $a \cong b$, holds if and only if $a \lesssim b$ and $b \lesssim a$.

The denotational semantics of terms of the imperative object calculus is extended to runtime values, stacks and heaps in the evident way. The definitions of $[\![v]\!] \in \mathsf{Val}$, $[\![\boldsymbol{\rho}]\!] \in \mathsf{Env}$ and $[\![\boldsymbol{\sigma}]\!] \in \mathsf{St}$ can be found in Table 3.9.

One direction of the agreement between operational and denotational semantics is easy to prove.

Lemma 3.3.7 (Soundness)**.** *If $\boldsymbol{\rho}; \boldsymbol{\sigma} \triangleright a \longmapsto v; \boldsymbol{\sigma}'$ then $[\![a]\!]\,[\![\boldsymbol{\rho}]\!]\,[\![\boldsymbol{\sigma}]\!] = \langle [\![v]\!], [\![\boldsymbol{\sigma}']\!] \rangle$.*

Table 3.9 *Semantics of runtime values, stacks and heaps*

$$[\![\mathsf{true}]\!] \stackrel{def}{=} true \qquad [\![\mathsf{false}]\!] \stackrel{def}{=} false \qquad [\![l]\!] \stackrel{def}{=} l$$

$$[\![\boldsymbol{\rho}]\!](x) \stackrel{def}{=} \begin{cases} [\![v]\!] & \text{if } \boldsymbol{\rho}(x) = v \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\boldsymbol{\sigma}]\!] \stackrel{def}{=} \{\!|l = o_l|\!\}_{l \in \mathsf{dom}(\boldsymbol{\sigma})} \quad \text{where } o_l \stackrel{def}{=} \langle \{\!|\mathsf{f}_i = [\![v_i]\!]|\!\}_{i\in I}, \{\!|\mathsf{m}_j = \lambda\langle l, \sigma\rangle.\, [\![b_j]\!]\,([\![\boldsymbol{\rho}_j]\!]\,[y_j := l])\sigma|\!\}_{j\in J}\rangle$$
$$\text{for } \boldsymbol{\sigma}.l = [\mathsf{f}_i = v_i, \mathsf{m}_j = \langle \varsigma(y_j) b_j, \boldsymbol{\rho}_j\rangle]_{i\in I, j\in J}$$

Table 3.10 *Semantics of contexts*

$$[\![C]\!]^{ctxt} : (\mathsf{Env} \to \mathsf{St} \rightharpoonup \mathsf{Val} \times \mathsf{St}) \to \mathsf{Env} \to \mathsf{St} \rightharpoonup (\mathsf{Val} + \{\mathsf{error}\}) \times \mathsf{St}$$

$$[\![\bullet]\!]^{ctxt} h\rho\sigma \stackrel{def}{=} (h\rho)\sigma$$

$$[\![\mathsf{if}\ x\ \mathsf{then}\ C\ \mathsf{else}\ b]\!]^{ctxt} h\rho\sigma \stackrel{def}{=} \begin{cases} [\![C]\!]^{ctxt} h\rho\sigma' & \text{if } [\![x]\!]\rho\sigma = \langle true, \sigma'\rangle \\ [\![b]\!]\rho\sigma' & \text{if } [\![x]\!]\rho\sigma = \langle false, \sigma'\rangle \\ \langle \mathsf{error}, \sigma'\rangle & \text{if } [\![x]\!]\rho\sigma = \langle v, \sigma'\rangle \text{ for } v \notin \mathsf{BVal} \end{cases}$$

Proof. By a straightforward induction on the derivation of $\boldsymbol{\rho}; \boldsymbol{\sigma} \rhd a \longmapsto v; \boldsymbol{\sigma}'$. □

Two cases of the semantics $[\![C]\!]^{ctxt}$ of contexts $C \in Ctxt$ are defined in Table 3.10; the remaining ones are similarly straightforward and omitted. This semantics is compositional:

Lemma 3.3.8 (Compositionality)**.** *For all $C \in Ctxt$ and $a \in Ob$, $[\![C[a]]\!] = [\![C]\!]^{ctxt}([\![a]\!])$.*

Proof. By induction on C. □

Next we consider *formal approximation relations*,

- $\lhd_{St} \subseteq \mathsf{St} \times Store$, stating that (i) all the methods of all the objects in a denotational store σ and operational heap $\boldsymbol{\sigma}$ are related by $\lhd$, and (ii) all the values of the fields are equal; and
- $\lhd \subseteq \mathsf{Cl} \times Clos$, where $h \lhd \langle \varsigma(y)b, \boldsymbol{\rho}\rangle$ means that termination of h implies termination (with equal result) of $\langle \varsigma(y)b, \boldsymbol{\rho}\rangle$ when run in stores related by $\lhd_{St}$, and moreover the $\lhd_{St}$ relation is preserved.

The formal definition of these relations is provided in Table 3.11. Because $\lhd_{St}$ is defined in terms of $\lhd$, and there is a negative occurrence of $\lhd_{St}$ in the right hand side for $\lhd$, we must show well-definedness of the formal approximation relations. This is done in the following by appealing to the existence theorem of Chapter 2, Theorem 2.5.2.

For a cpo D let $\mathcal{R}(D)$ consist of all relations $R \subseteq D \times Store$ such that for each $\boldsymbol{\sigma} \in Store$ the set $\{d \in D \mid \langle d, \boldsymbol{\sigma}\rangle \in R\}$ forms an admissible subset of D, i.e., for which

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \ \wedge\ \forall i \in \mathbb{N}.\, \langle d_i, \boldsymbol{\sigma}\rangle \in R \quad \Longrightarrow \quad \langle \bigsqcup_{i\in\mathbb{N}} d_i, \boldsymbol{\sigma}\rangle \in R$$

Table 3.11 *Formal Approximation*

$$\sigma \lhd_{St} \boldsymbol{\sigma} \quad \stackrel{def}{\Longleftrightarrow} \quad \left\{ \begin{array}{ll} \mathsf{dom}(\sigma) = \mathsf{dom}(\boldsymbol{\sigma}) & \\ \wedge\ \mathsf{dom}(\sigma.l) = \mathsf{dom}(\boldsymbol{\sigma}.l) & \forall l \in \mathsf{dom}(\sigma) \\ \quad \wedge\ \sigma.l.\mathsf{m} \lhd \boldsymbol{\sigma}.l.\mathsf{m} & \forall \mathsf{m} \in \mathsf{dom}(\sigma.l) \\ \quad \wedge\ \sigma.l.\mathsf{f} = [\![\boldsymbol{\sigma}.l.\mathsf{f}]\!] & \forall \mathsf{f} \in \mathsf{dom}(\sigma.l) \end{array} \right.$$

$$m \lhd \langle \varsigma(y)b, \boldsymbol{\rho} \rangle \quad \stackrel{def}{\Longleftrightarrow} \quad \left\{ \begin{array}{l} \forall \sigma, \sigma' \in \mathsf{St}\ \forall \boldsymbol{\sigma} \in \mathit{Store}\ \forall l \in \mathsf{Loc}\ \forall v \in \mathsf{Val}. \\ (\sigma \lhd_{St} \boldsymbol{\sigma}\ \wedge\ m(l,\sigma) = \langle v, \sigma' \rangle \quad \Longrightarrow \\ \quad \exists \boldsymbol{\sigma}' \in \mathit{Store}\ \exists v' \in \mathit{Val}. \\ \quad\quad (\boldsymbol{\rho}, y{\mapsto}l); \boldsymbol{\sigma} \rhd b \longmapsto v'; \boldsymbol{\sigma}'\ \wedge\ \sigma' \lhd_{St} \boldsymbol{\sigma}'\ \wedge\ v = [\![v']\!]) \end{array} \right.$$

Table 3.12 *Functionals* $\Phi_{\lhd}$ *and* $\Phi_{\lhd_{St}}$

$$\Phi_{\lhd}(R,S) \stackrel{def}{=} \left\{ \langle h, \langle \varsigma(y)b, \boldsymbol{\rho} \rangle \rangle \;\middle|\; \begin{array}{l} \forall \sigma, \sigma' \in D\ \forall \boldsymbol{\sigma} \in \mathit{Store}\ \forall l \in \mathsf{Loc}\ \forall v \in \mathsf{Val}. \\ \langle \sigma, \boldsymbol{\sigma} \rangle \in R\ \wedge\ h(l,\sigma) = \langle v, \sigma' \rangle \quad \Longrightarrow \\ \quad \exists v' \in \mathit{Val} \exists \boldsymbol{\sigma}' \in \mathit{Store}. \\ \quad\quad (\boldsymbol{\rho}, y{\mapsto}l); \boldsymbol{\sigma} \rhd b \longmapsto v'; \boldsymbol{\sigma}'\ \wedge\ \langle \sigma', \boldsymbol{\sigma}' \rangle \in S\ \wedge\ v = [\![v']\!] \end{array} \right\}$$

$$\Phi_{\lhd_{St}}(R,S) \stackrel{def}{=} \left\{ \langle \sigma, \boldsymbol{\sigma} \rangle \;\middle|\; \begin{array}{ll} \mathsf{dom}(\sigma) = \mathsf{dom}(\boldsymbol{\sigma}) & \\ \wedge\ \mathsf{dom}(\sigma.l) = \mathsf{dom}(\boldsymbol{\sigma}.l) & \forall l \in \mathsf{dom}(\sigma) \\ \wedge\ \langle \sigma.l.\mathsf{m}, \boldsymbol{\sigma}.l.\mathsf{m} \rangle \in \Phi_{\lhd}(R,S) & \forall \mathsf{m} \in \mathsf{dom}(\sigma.l) \\ \wedge\ \sigma.l.\mathsf{f} = [\![\boldsymbol{\sigma}.l.\mathsf{f}]\!] & \forall \mathsf{f} \in \mathsf{dom}(\sigma.l) \end{array} \right\}$$

holds. Further, we define for partial continuous $f : D \rightharpoonup E$ and $R \in \mathcal{R}(D)$ and $S \in \mathcal{R}(E)$,

$$f : R \subset S \quad \stackrel{def}{\Longleftrightarrow} \quad \forall \langle d, \boldsymbol{\sigma} \rangle \in R.\, f(d)\!\downarrow \quad \Longrightarrow \quad \langle f(d), \boldsymbol{\sigma} \rangle \in S$$

According to (Pitts 1996) this makes $\mathcal{R}$ a normal relational structure on **pCpo**, with intersections given by set-theoretic intersection and inverse images given by

$$f^*S = \{ \langle d, \boldsymbol{\sigma} \rangle \in D \times \mathit{Store} \mid f(d)\!\downarrow\ \wedge \langle f(d), \boldsymbol{\sigma} \rangle \in S \}$$

For all E and $S \in \mathcal{R}(E)$, $[R,S]$ contains $\perp$ and is closed under taking least upper bounds, thus all $S \in \mathcal{R}(E)$ are admissible in the sense of Section 2.5.1.

Table 3.12 defines a functional $\Phi_{\lhd_{St}}$ on this relational structure, such that

$$R \in \mathcal{R}(D)\ \wedge\ S \in \mathcal{R}(E) \quad \Longrightarrow \quad \Phi_{\lhd_{St}}(R,S) \in \mathcal{R}(F_{\mathit{Store}}(D,E)) \tag{3.8}$$

Verifying that the set $\{\sigma \mid \langle \sigma, \boldsymbol{\sigma} \rangle \in \Phi_{\lhd_{St}}(R,S)\} \subseteq F_{\mathit{Store}}(D,E)$ is admissible relies on the operational semantics being deterministic, because of the existential quantifiers in the definition of $\Phi_{\lhd}$. In the statement of the operational semantics on page 47 we glossed over the issue of how to choose fresh locations, used in (Red Obj) and (Red Copy). Thus the semantics as given is *not* deterministic. However, this is the only reason for indeterminism and the problem can be rectified by specifying the allocation mechanism more

precisely: For instance, a common solution is to let Loc be a well-ordered set such as the natural numbers, and choose the next location according to

$$\mathsf{next}(\boldsymbol{\sigma}) \stackrel{def}{=} \min\{n \mid n \notin \mathsf{dom}(\boldsymbol{\sigma})\}$$

depending on the locations already present in a heap. The same observation applies to the denotational semantics. We omit the details and will be similarly sloppy in our treatment of allocation throughout this thesis. A neater and more formal development should be possible using the FM cpos of (Shinwell 2005; Shinwell and Pitts 2005), but we have not investigated this further.

Next we show that for all $f : D' \rightharpoonup D$, $g : E \rightharpoonup E'$, $R \in \mathcal{R}(D)$, $R' \in \mathcal{R}(D')$, $S \in \mathcal{R}(E)$ and $S' \in \mathcal{R}(E')$,

$$f : R' \subset R \;\wedge\; g : S \subset S' \quad\Longrightarrow\quad F_{Store}(f,g) : \Phi_{\triangleleft St}(R,S) \subset \Phi_{\triangleleft St}(R',S') \tag{3.9}$$

Suppose $\langle \sigma, \boldsymbol{\sigma}\rangle \in \Phi_{\triangleleft St}(R,S)$ and $F_{Store}(f,g)(\sigma)\downarrow$. We have to show $\langle F_{Store}(f,g)(\sigma), \boldsymbol{\sigma}\rangle \in \Phi_{\triangleleft St}(R',S')$. By definition of F_{Store} in (3.6) on page 48,

$$\begin{aligned} &F_{Store}(f,g)(\sigma).l.\mathsf{f} = \sigma.l.\mathsf{f} && \forall l \in \mathsf{Loc}\ \forall \mathsf{f} \in \mathcal{F} \\ &F_{Store}(f,g)(\sigma).l.\mathsf{m}(l',\sigma') = \langle \nu, g(\sigma'')\rangle && \forall l,l' \in \mathsf{Loc}\ \forall \mathsf{m} \in \mathcal{M}\ \forall \sigma' \in \mathsf{St} \end{aligned}$$

where $\langle \nu, \sigma''\rangle \stackrel{def}{=} \sigma.l.\mathsf{m}(l', f(\sigma'))$. In particular,

$$\begin{aligned} &\mathsf{dom}(F_{Store}(f,g)(\sigma)) = \mathsf{dom}(\sigma) = \mathsf{dom}(\boldsymbol{\sigma}) && \\ &\mathsf{dom}(F_{Store}(f,g)(\sigma).l) = \mathsf{dom}(\sigma.l) = \mathsf{dom}(\boldsymbol{\sigma}.l) && \forall l \in \mathsf{dom}(\sigma) \\ &F_{Store}(f,g)(\sigma).l.\mathsf{f} = \sigma.l.\mathsf{f} = \boldsymbol{\sigma}.l.\mathsf{f} && \forall l \in \mathsf{dom}(\sigma)\ \forall \mathsf{f} \in \mathsf{dom}(\sigma.l) \end{aligned}$$

By definition of $\Phi_{\triangleleft St}$ it only remains to show $\langle F_{Store}(f,g)(\sigma).l.\mathsf{m}, \boldsymbol{\sigma}.l.\mathsf{m}\rangle \in \Phi_{\triangleleft}(R',S')$ for all $l \in \mathsf{dom}(\sigma)$ and $\mathsf{m} \in \mathsf{dom}(\sigma.l)$. So let $\langle \sigma', \boldsymbol{\sigma}'\rangle \in R'$, let $l' \in \mathsf{Loc}$ and suppose

$$F_{Store}(f,g)(\sigma).l.\mathsf{m}(l',\sigma') = \langle \nu, \sigma''\rangle$$

Therefore, by the observation made above, there exists $\hat{\sigma} \in \mathsf{St}$ such that $\sigma.l.\mathsf{m}(l',\sigma') = \langle \nu, \hat{\sigma}\rangle$ and $\sigma'' = g(\hat{\sigma})$. By assumption, $f : R' \subset R$, so that $\langle f(\sigma'), \boldsymbol{\sigma}'\rangle \in R$. Hence, by the assumption $\langle \sigma, \boldsymbol{\sigma}\rangle \in \Phi_{\triangleleft St}(R,S)$, if $\sigma.l.\mathsf{m} = \langle \varsigma(x)a, \boldsymbol{\rho}\rangle$ there exist $\nu' \in Val$ and $\boldsymbol{\sigma}'' \in Store$ such that

$$(\boldsymbol{\rho}, x{\mapsto}l'); \boldsymbol{\sigma}' \rhd a \longmapsto \nu'; \boldsymbol{\sigma}''$$

where $\langle \hat{\sigma}, \boldsymbol{\sigma}''\rangle \in S$ and $\nu = [\![\nu']\!]$. Finally, by assumption $g : S \subset S'$, we also have $\langle \sigma'', \boldsymbol{\sigma}''\rangle = \langle g(\hat{\sigma}), \boldsymbol{\sigma}''\rangle \in S$. Thus $\langle F_{Store}(f,g)(\sigma).l.\mathsf{m}, \langle \varsigma(x)a, \boldsymbol{\rho}\rangle\rangle \in \Phi_{\triangleleft}(R',S')$ and we have proved (3.9).

Properties (3.8) and (3.9) express that $\Phi_{\triangleleft St}$ is an admissible action of the bifunctor F_{Store}:

Lemma 3.3.9 (Admissible Action)**.** *The functor* $F_{Store} : \mathbf{pCpo}^{op} \times \mathbf{pCpo} \longrightarrow \mathbf{pCpo}$ *has an admissible action on* $\mathcal{R}$*, given by* $\Phi_{\triangleleft St}$*.*

Thus by Theorem 2.5.2 the invariant $\mathcal{R}$-relation for the action of F_{Store} exists, which we denote by $\lhd_{St}$:

$$\lhd_{St} = \Phi_{\lhd_{St}}(\lhd_{St}, \lhd_{St}) \in \mathcal{R}(\mathsf{St})$$

We can finally define $\lhd \stackrel{def}{=} \Phi_{\lhd}(\lhd_{St}, \lhd_{St})$ to obtain the required formal approximation relation between denotational and operational method closures.

Lemma 3.3.10. *Let $a \in Ob$, $\boldsymbol{\rho} \in Env$ and $\boldsymbol{\sigma} \in Store$. Then the following hold,*

- $\lambda\langle l, \sigma\rangle.\, [\![a]\!]\,([\![\boldsymbol{\rho}]\!]\,[x := l])\sigma \lhd \langle \varsigma(x)a, \boldsymbol{\rho}\rangle$*; and*
- $[\![\boldsymbol{\sigma}]\!] \lhd_{St} \boldsymbol{\sigma}$*.*

Proof. The first part is by (a tedious) induction on a and exploiting the defining property of the $\lhd$ relation. The second part then follows from the definitions of $[\![\boldsymbol{\sigma}]\!]$ and $\lhd_{St}$. □

Lemma 3.3.11. *For all programs $a \in Prog$ and all $\boldsymbol{\sigma} \in Store$, if $[\![a]\!]\,[\![\boldsymbol{\sigma}]\!] = \langle v, \boldsymbol{\sigma}'\rangle$ such that $v \neq \mathsf{error}$, then $\boldsymbol{\sigma} \rhd a \Downarrow$.*

Proof. By Lemma 3.3.10, $\lambda\langle l, \sigma\rangle.\, [\![a]\!]\,([\![\boldsymbol{\rho}]\!]\,[x := l])\sigma \lhd \langle \varsigma(x)a, \boldsymbol{\rho}\rangle$ and $[\![\boldsymbol{\sigma}]\!] \lhd_{St} \boldsymbol{\sigma}$. Since $\mathsf{fv}(a) = \emptyset$, for any $\boldsymbol{\rho} \in Env$ and $l \in \mathsf{Loc}$

$$\langle v, \sigma'\rangle = [\![a]\!]\,[\![\boldsymbol{\sigma}]\!] = [\![a]\!]\,([\![\boldsymbol{\rho}]\!]\,[x := l])\,[\![\boldsymbol{\sigma}]\!]$$

Hence by definition of $\lhd$, there exist $v' \in Val$ and $\boldsymbol{\sigma}' \in Store$ such that $(\boldsymbol{\rho}, x \mapsto l); \boldsymbol{\sigma} \rhd a \longmapsto v'; \boldsymbol{\sigma}'$ and $v = [\![v']\!]$ and $\sigma' \lhd_{St} \boldsymbol{\sigma}'$. In particular, $\boldsymbol{\sigma} \rhd a \Downarrow$ since $a \in Prog$ entails $\boldsymbol{\sigma} \rhd a \longmapsto v'; \boldsymbol{\sigma}'$ (without the stack) too. □

Theorem 3.3.12 (Adequacy)**.** *For all $a, b \in Ob$, if $[\![a]\!] \sqsubseteq [\![b]\!]$ then $a \precsim b$. In particular, $[\![a]\!] = [\![b]\!]$ implies $a \cong b$.*

Proof. Let C be any context such that both $C[a]$ and $C[b]$ are closed, let $\boldsymbol{\sigma} \in Store$ a store, and suppose $\boldsymbol{\sigma} \rhd C[a] \Downarrow$, i.e.,

$$\boldsymbol{\sigma} \rhd C[a] \longmapsto v; \boldsymbol{\sigma}'$$

for some $v \in Val$ and $\boldsymbol{\sigma}' \in Store$. Let $\sigma \stackrel{def}{=} [\![\boldsymbol{\sigma}]\!]$. Then

$$\begin{aligned}
[\![C[b]]\!]\,\sigma &= [\![C]\!]^{ctxt}([\![b]\!])\sigma && \text{by (Compositionality), Lemma 3.3.8}\\
&\sqsupseteq [\![C]\!]^{ctxt}([\![a]\!])\sigma && \text{since } [\![a]\!] \sqsubseteq [\![b]\!]\\
&= [\![C[a]]\!]\,\sigma && \text{by (Compositionality), Lemma 3.3.8}\\
&= \langle [\![v]\!], [\![\boldsymbol{\sigma}']\!]\rangle && \text{by (Soundness), Lemma 3.3.7}
\end{aligned}$$

Thus there exist $\sigma' \in \mathsf{St}$ and $v' \in \mathsf{Val}$ such that $[\![C[b]]\!]\,\sigma = \langle v', \sigma'\rangle$. In particular, $v' \neq \mathsf{error}$. By Lemma 3.3.11, $\boldsymbol{\sigma} \rhd C[b] \Downarrow$, and we have shown $a \precsim b$. □

Table 3.13 *Syntax of a class-based language. Types and terms*

A, B	::=	X | bool | ...	type variable, base types
	|	$\mathsf{Object}(X)\,[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j]_{i,j}$	(recursive) object type
	|	$\mathsf{Class}(A)$	class type
a, b	::=	x	variable
	|	true | ...	expressions, conditional, let,...
	|	$a.\mathsf{f}$ | $a.\mathsf{m}$	field selection, method invocation
	|	new $c\,(a_1, \ldots, a_n)$	instance creation
c, d	::=	root	root class
	|	subclass of $c{:}C$ with $(x{:}A)$	class definition
		$(\mathsf{f}_i)_i$	set of additional fields, $i \in I$
		$(\mathsf{m}_j = b_j)_j$	set of additional methods, $j \in J$
		$(\mathsf{override}\ \mathsf{m}_k = b_k)_k$	set of overridden methods, $k \in K$
		end	

3.4 Modelling Class-based Languages

Many formalisations of class-based languages have appeared in the literature. Widely cited examples are FEATHERWEIGHT JAVA (Igarashi, Pierce, and Wadler 2001; Bracha, Odersky, Stoutamire, and Wadler 1998; Igarashi and Pierce 2000), KOOL (Castagna 1997) and the SOOL family of languages of (Bruce 2002; Bruce 1994). Here, for simplicity, we present a variant of the class-based language from Chapter 12 of Abadi and Cardelli's (1996) book: It can be quite straightforwardly mapped into the object calculus of the previous section, and class-based reasoning reduces to object-based reasoning principles.

The ingredients of this translation are the following. Classes are viewed as object generators, a view that is well in accordance with the theoretical literature on the subject (Cook 1989; Cook and Palsberg 1994; Kamin and Reddy 1994; Bruce 1994; Abadi and Cardelli 1996; Bono, Patel, Shmatikov, and Mitchell 1999; Boudol 2004). More precisely,

- a class is a record of *pre-methods*, i.e., functions abstracted on the self-parameter;
- inheritance is extension of this record, possibly replacing some of the pre-methods in the modelling of method redefinition ("override");
- object creation ("new") is the creation of an object from the record of pre-methods, obtained by binding the self-parameter.

3.4.1 A Class-based Language

Table 3.13 collects the syntax of types and terms of a simple class-based language. Types include base types, (recursive) object types and class types. In order to distinguish between object and class types, we follow Abadi and Cardelli in writing object types now as $\mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j\right]_{i,j}$ instead of $\mu(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j\right]_{i,j}$. If A is an object type, then

$\mathsf{Class}(A)$ is a well-formed class type:

$$\frac{A \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j\right]_{i,j} \qquad \Gamma \rhd A}{\Gamma \rhd \mathsf{Class}(A)}$$

It is the type of classes whose instance objects have type A. The class type Root is defined by

$$\mathsf{Root} \stackrel{def}{=} \mathsf{Class}(\mathsf{Object}(X)\ [\,])$$

Much emphasis in class-based languages is put on *inheritance* (Taivalsaari 1996). Informally, a class consists of a set of pre-methods, that is, methods that are abstracted on the self parameter. Viewing inheritance as pre-method reuse, one defines an inheritance relation on class types based on the subtype relation on the underlying object types,

$$\mathsf{Class}(A) \text{ may inherit from } \mathsf{Class}(B) \quad \stackrel{def}{\iff} \quad \rhd A \preceq B$$

meaning that a class $c{:}\mathsf{Class}(A)$ may be defined with reference to some class $d{:}\mathsf{Class}(B)$.

The base class, root, contains no pre-methods. All other classes must be defined using the subclass construct. Given a class $c{:}\mathsf{Class}(A)$ then

$$\mathsf{subclass\ of}\ c{:}\mathsf{Class}(A)\ \mathsf{with}\ (x{:}B)\ \mathsf{f}_i,\ \mathsf{m}_j = b_j,\ \mathsf{override}\ \mathsf{m}_k = b_k\ \mathsf{end}$$

constructs a new class $d{:}\mathsf{Class}(B)$. The class d consists of all the pre-methods of c except for those m_k that have been replaced by a new definition b_k listed in the override clause, and extends this set with pre-methods $\mathsf{m}_j = b_j$. Instances of this class have the fields of instances of c, and additionally fields f_i. The variable $x{:}B$ is the self variable for all the methods in this class definition. The type system will guarantee that B is an object type such that all pre-methods may be soundly applied to the instance objects.

The typing rules for subtyping and terms remain much as before; Table 3.14 contains the new additional rules for classes. Note that there is no subtyping on class types as there appears to be no reasonable definition that is connected to the more fundamental may-inherit-from relation between class types.

The rule for the instance creation new ensures that the initialisation values for the fields have the right type. The rule for the base class has the obvious form. The type rule for subclassing is rather complex. It ensures that the instance type of the subclass is a subtype of the instance type of the superclass, i.e., that the subclass may indeed inherit from the superclass. In addition, a certain well-formedness is guaranteed by (1) ruling out any name clashes between fields inherited from the superclass and those defined for the subclass, (2) ruling out any name clashes between methods inherited from the superclass and those additionally defined for the subclass, and (3) checking that only methods defined in the superclass are overridden.

3.4.2 Classes as Objects

We summarise the translation $(-)^\circ$ into the object calculus proposed by Abadi and Cardelli next. The intuition is that a class corresponds to the record of its pre-methods, enriched

Table 3.14 *Typing of terms and classes*

(TERM NEW)
$$\frac{\begin{array}{c} A \equiv \mathsf{Object}(X)[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j]_{i\in I, j\in J} \\ \Gamma \rhd c : \mathsf{Class}(A) \\ \Gamma \rhd x_i : A_i \qquad \forall i \in I = \{1,\ldots,n\} \end{array}}{\Gamma \rhd \mathsf{new}\ c(x_1,\ldots,x_n) : A}$$

(TERM ROOT)
$$\frac{}{\Gamma \rhd \mathsf{root} : \mathsf{Root}}$$

(TERM SUBCLASS)
$$\frac{\begin{array}{c} A \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B'_j\right]_{i\in I', j\in J'} \\ B \equiv \mathsf{Object}(X)[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j]_{i\in I\uplus I', j\in J\uplus J'} \\ \Gamma \rhd c : \mathsf{Class}(A) \qquad \Gamma \rhd B \preceq A \qquad K \subseteq J' \\ \Gamma \rhd B_j[B/X] \preceq B'_j[B/X] \qquad \forall j \in J' - K \\ \Gamma, x{:}B \rhd b_l : B_l[B/X] \qquad \forall l \in J \cup K \end{array}}{\begin{array}{ll} \Gamma \rhd & \mathsf{subclass\ of}\ c{:}\mathsf{Class}(A)\ \mathsf{with}\ (x{:}B) \qquad\qquad : \mathsf{Class}(B) \\ & \quad (\mathsf{f}_i)_{i\in I}\ (\mathsf{m}_j = b_j)_{j\in J}\ (\mathsf{override}\ \mathsf{m}_k = b_k)_{k\in K} \\ & \quad \mathsf{end} \end{array}}$$

Table 3.15 *Translation of types*

$$\begin{array}{lcl} X^\circ & \stackrel{def}{=} & X \\ \mathsf{bool}^\circ & \stackrel{def}{=} & \mathsf{bool} \\ \mathsf{Object}(X)[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j]_{i,j}{}^\circ & & \\ \qquad & \stackrel{def}{=} & \mu(X)[\mathsf{f}_i{:}A_i^\circ,\ \mathsf{m}_j{:}B_j^\circ]_{i,j} \\ \mathsf{Class}(A)^\circ & \stackrel{def}{=} & \left[\mathit{new}{:}(A_1^\circ \times \cdots \times A_n^\circ) \Rightarrow A^\circ, \mathsf{m}_j{:}A^\circ \Rightarrow B_j^\circ[A^\circ/X]\right]_j \\ \multicolumn{3}{l}{\text{where } A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j]_{i\in I, j\in J},\ I = \{1,\ldots,n\}} \end{array}$$

with a method *new* that assembles the pre-methods into objects. Therefore instance creation $\mathsf{new}\ c\,(\ldots)$ corresponds directly to the method invocation $c^\circ.\mathit{new}(\ldots)$. Table 3.15 contains the translation of types that directly implements this view.

Translation of Terms

Table 3.16 summarises the translation of terms into the object calculus. The base class root translates to an object with a method *new* that generates empty objects. A subclass definition translates to an object that contains the set of new and overridden pre-methods, m_j for $j \in J' \uplus K$, as well as the inherited pre-methods that are not overridden, m_j for $j \in J - K$. The latter are implemented by referring to the superclass c, as $\mathsf{m}_j = c^\circ.\mathsf{m}_j$. Finally, the method *new* assembles all these pre-methods in an object and binds their self-parameter by replacing the abstraction with a ς binder. The fields are initialised with the given arguments.

Functions and Pre-Methods

We have not yet discussed how to encode functions (and methods with input parameters) in the object calculus, which is necessary in order to make precise the translation of

Table 3.16 *Translation of terms*

$$
\begin{array}{lcl}
(\mathsf{new}\ c\ (x_1,\ldots,x_n))^\circ & \stackrel{def}{=} & c^\circ.new(x_1,\ldots,x_n) \\
\mathsf{root}^\circ & \stackrel{def}{=} & [new = \varsigma(y)[\,]] \\
(\mathsf{subclass\ of}\ c{:}\mathsf{Class}(A)\ \mathsf{with}\ (x{:}B)\ \mathsf{f}_i,\ \mathsf{m}_j = b_j,\ \mathsf{override}\ \mathsf{m}_k = b_k\ \mathsf{end})^\circ & & \\
 & \stackrel{def}{=} & \mathsf{let}\ y{=}c^\circ\ \mathsf{in} \\
 & & \left[\begin{array}{l} new = \varsigma(y)\lambda\langle x_1,\ldots,x_n\rangle[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(z)y.\mathsf{m}_j(z)]_{i\in I\uplus I',j\in J\uplus J'} \\ (\mathsf{m}_j = y.\mathsf{m}_j)_{j\in J-K} \\ (\mathsf{m}_l = \lambda x.b_l^\circ)_{l\in J'\uplus K} \end{array}\right]
\end{array}
$$

where $I \uplus I' = \{1,\ldots,n\}$

$A \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B'_j\right]_{i\in I',j\in J'}$

$B \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}B_j\right]_{i\in I\uplus I',j\in J\uplus J'}$

classes as objects containing pre-methods and a constructor method. The key idea is that supplying the input proceeds via a new field arg that is updated with the actual parameter prior to evaluating the function or method. Thus, a function type $A \Rightarrow B$ corresponds to an object type $(A \Rightarrow B)^* \stackrel{def}{=} [\mathsf{arg} : A^*, \mathsf{eval} : B^*]$, an abstraction $\lambda x.e$ corresponds to an object of this type

$$(\lambda x.e)^* \stackrel{def}{=} [\mathsf{arg} = \ldots, \mathsf{eval} = \varsigma(y)e^*[y.\mathsf{arg}/x]]$$

(the arg field is initially set to an arbitrary value of type A) and application $a(b)$ is

$$(a(b))^* \stackrel{def}{=} (\mathsf{clone}(a^*).\mathsf{arg} := b^*).\mathsf{eval}$$

The cloning of the object is necessary to simulate a proper call-stack, otherwise recursive calls would interfere. An alternative method to achieve this effect would be to explicitly restore arg to its original value after the call:

$$(a(b))^* \stackrel{def}{=} \mathsf{let}\ f{=}(a^*)\ \mathsf{in}\ \mathsf{let}\ y{=}f.\mathsf{arg}\ \mathsf{in}\ \mathsf{let}\ r{=}(f.\mathsf{arg} := b^*).\mathsf{eval}\ \mathsf{in}\ (f.\mathsf{arg} := y;\ r)$$

Of course it is also possible to directly extend the syntax and semantics (Section 3.3) of the object calculus, introducing cases for methods with parameters.

3.4.3 Benefits of Class Objects

We believe there are the following advantages for modelling classes in this way. Firstly, from a pragmatic point of view, this allows us to directly transfer results and techniques for reasoning about objects to reasoning about classes. In particular, Abadi and Leino's (1997, 2004) logic gives rise to a logic for class-based languages in this way. Secondly, we believe that the translation leads to a semantic model of class-based languages that does not suffer from the short-comings of the usual closed-world assumptions: The subsequent addition of further subclasses poses no problems with respect to modularity. Moreover, we note that the translation works just as well without distinguishing the syntactic categories $a, b, \ldots$ of terms and $c, d, \ldots$ of class expressions. By equating these,

class expressions become first-class entities of the language. In particular this directly allows for modelling of inner classes and methods returning dynamically constructed classes.

Part II

On Abadi and Leino's Logic

Chapter 4

Abadi and Leino's Logic of Objects and its Denotational Semantics

---

Abadi-Leino Logic (Abadi and Leino 1997, 2004) is a Hoare-calculus style logic for the simple imperative and object-based language introduced in Chapter 3. As can be seen from the semantics (both operational and denotational) every object comes with its own method suite. Thus methods need to reside in the store, i.e., programs use a higher-order store. In the logic one can prove properties of such programs.

In this part of the thesis, we present a new soundness proof for this logic using the denotational semantics of Section 3.3.3: The object specifications of the logic 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. We also extend the logic with a notion of recursive specification.

4.1 Introduction

For object-oriented languages several formal systems have been proposed, for instance (Abadi and Leino 2004; Hensel, Huisman, Jacobs, and Tews 1998; Jacobs and Poll 2001; Reddy 2002; Poetzsch-Heffter and Müller 1999; de Boer 1999; von Oheimb 2001; Reus, Wirsing, and Hennicker 2001). A "standard" comparable to the Hoare-calculus for imperative While-languages (Apt 1981) has not yet emerged. Moreover, 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 the work of Abadi and Leino (1997, 2004) where a logic for an object-based language is introduced that is derived from the imperative object calculus with first-order types, $\mathbf{imp}\varsigma$, (Abadi and Cardelli 1996). 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, 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

$$A \quad \equiv \quad [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}\, T_j]_{i\in I, j\in J}$$

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. Just as a method $\mathsf{m}_j = \varsigma(y_j)b_j$ can use the *self*-parameter y_j in its body b_j, 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:

$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}\, T_j]_{i\in I, j\in J} \\ \Gamma \vdash x_i{:}A_i{::}\ldots \quad \forall i \in I \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j \quad \forall j \in J \end{array}}{\Gamma \vdash \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j\right]_{i\in I, j\in J}{:}A{::}\ldots} \tag{4.1}$$

To make this more concrete let us consider an example. Suppose a is an object with a method that implements the factorial function,

$$\left[\begin{array}{l} \mathsf{arg} = 0, \\ \mathsf{fac} = \varsigma(y).\, \mathsf{if}\ y.\mathsf{arg} = 0\ \mathsf{then}\ 1 \\ \qquad\qquad \mathsf{else\ let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x-1; x \times (y.\mathsf{fac}) \end{array}\right] \tag{4.2}$$

Recall that in the object calculus of Section 3.3 methods do not have parameters, therefore the input is stored in the arg field of the object prior to calling fac (see also Section 3.4.2). Apart from this convention the program should be self-evident. The following specification

$$[\mathsf{arg} : \mathsf{int},\ \mathsf{fac} : \varsigma(y)\mathsf{int}{::}y.\mathsf{arg} \geq 0 \ \rightarrow\ \mathsf{result} = y.\mathsf{arg}!]$$

expresses (1) that the field arg contains an integer, (2) that the result of calling method fac is an integer (int is the result specification), and (3) that the execution of fac has the effect of computing y.arg! whenever y.arg ≥ 0 holds initially. In fact (2) and (3) are intended in the sense of partial correctness only, i.e., termination of fac is not required.

A closer look reveals that the transition specification $T \ \equiv\ y.\mathsf{arg}{\geq}0 \rightarrow \mathsf{result}{=}y.\mathsf{arg}!$ must be refined: Since the value of y.arg is decremented before each recursive call, in general it will be different in initial and result states. It is in fact not clear which stored value y.arg refers to in T. One standard approach to solve this problem is to use "primed" and "unprimed" variables to distinguish between these values (Lamport 1994). Abadi and Leino (2004) adopted a minor variation of this approach, explicating the selection of fields with respect to the initial state using $\mathsf{sel}_{pre}(y, \mathsf{arg})$, and $\mathsf{sel}_{post}(y, \mathsf{arg})$ with respect to the final state[1]. The treatment of predicates of the assertion language is likewise. Thus, a specification for the factorial program is

$$A \ \equiv\ [\mathsf{arg} : \mathsf{int},\ \mathsf{fac} : \varsigma(y)\mathsf{int}{::}\mathsf{sel}_{pre}(y, \mathsf{arg}) \geq 0 \ \rightarrow\ \mathsf{result} = \mathsf{sel}_{pre}(y, \mathsf{arg})!]$$

[1] Abadi and Leino (2004) use the notation $\grave{\sigma}(y, \mathsf{arg})$ and $\acute{\sigma}(y, \mathsf{arg})$

The proof rule (4.1) reduces a proof of $a : A$ to the (trivial) proof obligation $\vdash 0 : \mathsf{int}$, and to

$$y{:}A \vdash \mathsf{if}\ y.\mathsf{arg}{=}0\ \mathsf{then}\ 1\ \mathsf{else}\ \cdots : \mathsf{int} :: \mathsf{sel}_{pre}(y, \mathsf{arg}) \geq 0\ \rightarrow\ \mathsf{result} = \mathsf{sel}_{pre}(y, \mathsf{arg})! \quad (4.3)$$

The proof of (4.3) can now proceed by application of rules that are quite standard in Hoare calculus. The assumption $y{:}A$ is sufficient to reason about the recursive call, where we have to prove

$$y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{fac} : \mathsf{int} :: 0 \leq \mathsf{sel}_{pre}(y, \mathsf{arg})\ \rightarrow\ \mathsf{result} = (\mathsf{sel}_{pre}(y, \mathsf{arg}))!$$

Hence this example demonstrates that rule (4.1) in fact facilitates reasoning about recursion, and should give a first impression why establishing its soundness is difficult. The factorial example is proved in detail in Section 4.3.3, after presenting all the inference rules of Abadi-Leino logic.

In general, the result of a method may be another object, whose methods have to be specified in turn. That is, the result specifications may be object specifications themselves (instead of int in the factorial example), and moreover it may depend on the self parameter of the statically enclosing specification. An instance of this appears in the bank account of Example 3.3.2 whose specification may depend on the enclosing manager object. See Example 4.3.1 on page 68 below where this is carried out in more detail.

Abadi and Leino's logic is peculiar in another respect (although the property is well-known from the design from type systems): It is set up in a way such that specifications Γ that hold of objects in the context cannot be invalidated by proven code. In other words, although the correctness of an object may depend on the context (as in the bank account example) and the object may operate on the context (for instance by updating a field), this can not be done unrestrictedly: In the above factorial example (4.2), the method body may query and update the context $\Gamma \equiv y{:}A, x{:}\mathsf{int}$, as in the assignment $y.\mathsf{arg}{:=}x - 1$, but assigning a boolean or float to $y.\mathsf{arg}$ would invalidate Γ and is disallowed. Only programs preserving the context specification Γ in this sense can be proved correct in the logic.

4.2 Outline of Part II

In this and the following chapters we are going to present a new soundness proof for Abadi and Leino's logic, using an untyped denotational semantics of the language and the logic to define validity. Every program and every specification has 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, for instance, in LCF, a logic for functional programs (Paulson 1987).

The difficulty in our case is to establish predicates that correspond to the powerful reasoning principle for objects (4.1). Reus and Streicher (2002, 2004) have outlined how to use some techniques from domain theory (Pitts 1996) 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 their methods in the store. So in

general object specifications must depend on specifications of some of the other objects in the store, which gives rise to "store specifications". Indeed store specifications were already present in the operationally-based work of Abadi and Leino.

For the reasons given above, the development in this part is not simply an application of the ideas in (Reus and Streicher 2004). Much care is needed to establish the important invariance property of Abadi-Leino logic, namely that proved programs preserve store specifications. Our main achievement here is that we have successfully applied the ideas of (Reus and Streicher 2004) to the logic of (Abadi and Leino 2004) to obtain a soundness proof, with a view on *analysing this logic* and possibly *developing similar, but more powerful, program logics* as well.

Our soundness proof is not just "yet another proof" either. We consider it complementary to the one of Abadi and Leino (2004), which relies on the operational semantics of the object calculus and does not assign proper "meaning" to specifications. We believe the following reasons justify this claim:

- By using denotational semantics we can introduce a clear notion of *validity*, with no reference to *derivability*. This helps clarifying what the soundness proof is actually *stating* in the first place, see Section 7.1.1.
- We can extend the logic fairly easily, for instance by recursive specifications. This has been done for the Abadi-Leino logic in (Leino 1998) but for a slightly different programming language with *nominal subtyping* rather than the structural notion used here.
- Some essential restrictions of the logic are revealed and justified, by arguing that they correspond to *sufficient conditions* implying well-definedness of store specifications.
- 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 (Abadi and Leino 2004) transition specifications cannot talk about methods at all (see also Section 7.2).

We proceed as follows. In the next section, Abadi-Leino logic and the denotational semantics of its object specifications are presented. In Chapter 5 a discussion about store specifications and their semantics follows (Section 5.1); the main result is found in Section 5.2 where the logic is proved sound. Finally we show how recursive specifications can be introduced (Chapter 6), and we put our model in the wider context and discuss the possibility of further extensions (Chapter 7).

Remark 4.2.1. *Chapters 4–7 are based on the technical report (Reus and Schwinghammer 2004), an extended abstract of which appeared as (Reus and Schwinghammer 2005). Most results and proofs have had to be slightly adapted because here we are also considering the cloning construct. Recall that this feature requires (semantically speaking) a model where methods are maps not only from the store but also the self parameter: because the reference to self refers to different memory locations in the prototype and cloned object, respectively, self parameters cannot be frozen at creation time (see Section 3.2.3).*

4.3 Abadi-Leino Logic

We recall the logic of (Abadi and Leino 2004) next. A slightly different presentation can be found in (Tang and Hofmann 2002) where the proof system is given in a syntax-directed way.

4.3.1 Transition Relations and Specifications

Transition relations T correspond to the pre- and post-conditions of Hoare logic and allow to express state changes caused by computations. The syntax of transition relations is defined by the following grammar:

$$T ::= e_0 = e_1 \mid \mathsf{alloc}_{pre}(e) \mid \mathsf{alloc}_{post}(e) \mid \neg T \mid T_0 \wedge T_1 \mid \forall x.T$$
$$e ::= x \mid \mathsf{f} \mid \mathsf{result} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{sel}_{pre}(e_0, e_1) \mid \mathsf{sel}_{post}(e_0, e_1)$$

There is a constant for each field name $\mathsf{f} \in \mathcal{F}$ (which we just write f, too), and constants result, true and false. Intuitively, the function $\mathsf{sel}_{pre}(x, y)$ yields the value of field y of the object at location x before execution, provided this exists in the store, and is undefined otherwise. Correspondingly, $\mathsf{sel}_{post}(x, y)$ gives the value of field y after execution. The predicates $\mathsf{alloc}_{pre}(x)$ and $\mathsf{alloc}_{post}(x)$ are true if the location x is allocated before and after the execution, respectively, and false otherwise. The notions of free and bound variables of a transition relation T carry over directly from first-order logic. As usual, further logical connectives such as falsity and implication can be defined as abbreviations.

Specifications A combine transition relations for each method as well as the result types into a single specification for the whole object. They generalise the first-order types of (Abadi and Cardelli 1996; see Section 3.3.1), and are

$$A, B \in \mathit{Spec} ::= \mathsf{bool} \mid [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$$

In the case of an object specification, $A \equiv [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$, $\varsigma(y_j)$ binds the variable y_j in B_j and T_j. The sets $\mathsf{fv}(T)$ and $\mathsf{fv}(A)$ of free variables of T and A, resp., are defined in the obvious way. Specifications are identified up to renaming of bound variables and reordering of components. As for types, we define $\mathbf{1} \stackrel{def}{=} [\,]$ for the one-element specification.

Intuitively, true and false satisfy bool, and an object satisfies the object specification $A \equiv [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ 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. Corresponding to the fact that a method m_j can use the *self*-parameter y_j, in both T_j and B_j it is possible to refer to the ambient object y_j.

Well-formed Specification Contexts

In the following let Γ range over *specification contexts* $x_1{:}A_1, \ldots, x_n{:}A_n$. A specification context is *well-formed* if no variable x_i occurs more than once, and the free variables of A_k are contained in the set $\{x_1, \ldots, x_{k-1}\}$. As for the type contexts of the previous chapter, in writing $\Gamma, x{:}A$ we will always assume that x does not appear in Γ. Sometimes we write $\emptyset$

for the empty context. Given Γ, we write $[\Gamma]$ for the ordered list of variables occurring in Γ:

$$[x_1{:}A_1, \ldots, x_n{:}A_n] \stackrel{def}{=} x_1, \ldots, x_n$$

If clear from context, we use the notation $\overline{x}$ for a sequence $x_1, \ldots, x_n$, and similarly $\overline{x} : \overline{A}$ for $x_1{:}A_1, \ldots, x_n{:}A_n$. To make the notions of well-formed specifications and well-formed specification contexts formal, there are judgements for

- well-formed transition relations,

  $$x_1, \ldots, x_n \vdash T$$

  which holds if the free variables of transition relation T are contained in $x_1, \ldots, x_n$, i.e., $\mathsf{fv}(T) \subseteq \{x_1, \ldots, x_n\}$

- well-formed specifications, $\overline{x} \vdash A$,

  $$\text{(Spec Const)} \quad \frac{}{\overline{x} \vdash \mathsf{bool}}$$

  $$\text{(Spec Obj)} \quad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \overline{x} \vdash A_i \quad \forall i \in I \qquad \overline{x}, y_j \vdash B_j \qquad \overline{x}, y_j \vdash T_j \quad \forall j \in J \end{array}}{\overline{x} \vdash [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}}$$

- well-formed specification contexts, $\Gamma \vdash \mathsf{ok}$,

  $$\text{(Context Emp)} \quad \frac{}{\emptyset \vdash \mathsf{ok}}$$

  $$\text{(Context Spec)} \quad \frac{\Gamma \vdash \mathsf{ok} \quad [\Gamma] \vdash A \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x{:}A \vdash \mathsf{ok}}$$

If A is closed we may omit the empty context $\emptyset$, or even write A instead of $\vdash A$. Similarly for closed T. Essentially T – and therefore also A – depend on the list $[\Gamma]$ rather than Γ because the assertion language is *untyped* first-order logic.

Example 4.3.1. Table 4.3 on page 70 defines a few transition relations. Table 4.1 shows a specification for the bank accounts of Example 3.3.2 (page 42). Although we are using UML-like notation (Stevens and Pooley 2000), our diagram actually stands for individual objects, not classes – in fact there are no classes in the language. Observe how the transition relation T_{interest} depends not only on the self parameter y of the host object but also on the statically enclosing object x. □

4.3.2 Subspecifications and Proof System

Abadi and Leino generalised the notion of subtypes to a form of *subspecifications*, $\overline{x} \vdash A \prec: A'$. This relation is defined as the least reflexive and transitive relation on specifications closed under the inference rule of Table 4.2, where $\vdash_{\mathsf{fo}} \varphi$ denotes provability in first-order logic (in the theory with axioms for equality, and axioms stating that true, false and all $\mathsf{f} \in \mathcal{F}$ are distinct). Just as subtyping in the corresponding type system (Abadi and Cardelli 1996), the subspecification relation is covariant along method specifications and

Table 4.1 *An example of transition and result specifications*

$$T_{\mathtt{deposit}}(y) \equiv \exists z.z = \mathsf{sel}_{pre}(y, \mathtt{balance}) \wedge T_{\mathsf{upd}}(y, \mathtt{balance}, z + 10)$$

$$T_{\mathtt{withdraw}}(y) \equiv \exists z.z = \mathsf{sel}_{pre}(y, \mathtt{balance}) \wedge T_{\mathsf{upd}}(y, \mathtt{balance}, z - 10)$$

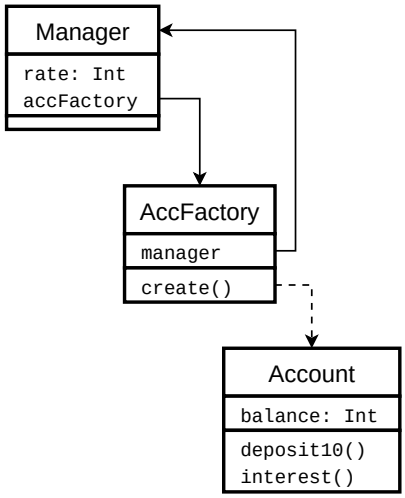
$$T_{\mathtt{interest}}(x, y) \equiv \exists z.z = \mathsf{sel}_{pre}(y, \mathtt{balance}) \wedge \exists m.m = \mathsf{sel}_{pre}(x, \mathtt{manager}) \wedge \exists r.r = \mathsf{sel}_{pre}(m, \mathtt{rate}) \wedge T_{\mathsf{upd}}(y, \mathtt{balance}, z * r/100)$$

$$T_{\mathtt{create}}(x) \equiv T_{\mathsf{obj}}(\mathtt{balance} = 0)$$

$$A_{\mathtt{Account}}(x) \equiv [\mathtt{balance} : \mathsf{int}, \mathtt{deposit10} : \varsigma(y)\mathbf{1} :: T_{\mathtt{deposit}}(y), \mathtt{withdraw10} : \varsigma(y)\mathbf{1} :: T_{\mathtt{withdraw}}(y), \mathtt{interest} : \varsigma(y)\mathbf{1} :: T_{\mathtt{interest}}(x, y)]$$

$$A_{\mathtt{AccFactory}} \equiv [\mathtt{manager} : [\mathtt{rate} : \mathsf{int}], \mathtt{create} : \varsigma(x)A_{\mathtt{Account}}(x) :: T_{\mathtt{create}}(x)]$$

$$A_{\mathtt{Manager}} \equiv [\mathtt{rate} : \mathsf{int}, \mathtt{accFactory} : A_{\mathtt{AccFactory}}]$$



transition relations, and invariant in field specifications (cf. rule (SUB OBJ) in Section 3.3.1). Observe that

$$\overline{x} \vdash A_1 \prec: A_2 \quad \Longrightarrow \quad \overline{x} \vdash A_i \tag{4.4}$$

for $i = 1, 2$.

In the logic, judgements of the form $\Gamma \vdash a{:}A{::}T$ can be derived, where Γ is a well-formed specification context, a is an object expression, A is a specification, and T is a transition relation. The rules guarantee that all the free variables of a, A and T appear in $[\Gamma]$. In the statement of the rules we use the transition relations defined in Table 4.3.

$T_{\mathsf{res}}(e)$ states that the result of a computation is e and the flat part of the store remains unchanged. $T_{\mathsf{obj}}(\mathsf{f}_i = x_i)$ describes the allocation of a new object in memory, which is initialised with field f_i set to x_i, and whose location is returned as result. $T_{\mathsf{upd}}(x, f, e)$ describes the effect on the store when updating field $x.f$. Note that in (Abadi and Leino 2004) T_{res} is called *Res* and T_{upd} is called *Update*. There is no abbreviation corresponding to T_{obj}.

There is one rule for each syntactic form of the language, and additionally a subsumption rule. In fact, the subsumption rule plays a central rôle, generalising the consequence rule of classical Hoare logic. The rules are given in Table 4.4.

As indicated before, one of the most interesting and powerful rules of the logic is the

Table 4.2 *The subspecification relation*

$$\text{(SUBSPEC OBJ)}\quad \frac{\begin{array}{llll} \overline{x} \vdash A_i & \forall i \in I & \overline{x}, y_j \vdash B_j & \forall j \in J \\ \overline{x}, y_j \vdash T_j & \forall j \in J & \overline{x}, y_j \vdash T'_j & \forall j \in J' \quad I' \subseteq I \\ \overline{x}, y_j \vdash B_j \prec: B'_j & \forall j \in J' & \vdash_{\mathsf{fo}} T_j \to T'_j & \forall j \in J' \quad J' \subseteq J \end{array}}{\overline{x} \vdash [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \prec: [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B'_j{::}T'_j]_{i\in I', j\in J'}}$$

Table 4.3 *Transition relations T_{res}, T_{obj} and T_{upd}*

$$T_{\mathsf{res}}(e) \stackrel{def}{=} \mathsf{result} = e \ \wedge\ \forall x\ \forall f.\ (\mathsf{alloc}_{pre}(x) \leftrightarrow \mathsf{alloc}_{post}(x) \ \wedge\ \mathsf{sel}_{pre}(x,f) = \mathsf{sel}_{post}(x,f))$$

$$T_{\mathsf{obj}}(\mathsf{f}_i = x_i)_{i\in I} \stackrel{def}{=} \neg\mathsf{alloc}_{pre}(\mathsf{result}) \ \wedge\ \mathsf{alloc}_{post}(\mathsf{result}) \ \wedge\ \textstyle\bigwedge_{i\in I} \mathsf{sel}_{post}(\mathsf{result}, \mathsf{f}_i) = x_i$$
$$\wedge\ \forall x\ \forall f.\ x \neq \mathsf{result} \to (\mathsf{alloc}_{pre}(x) \leftrightarrow \mathsf{alloc}_{post}(x) \ \wedge\ \mathsf{sel}_{pre}(x,f) = \mathsf{sel}_{post}(x,f))$$

$$T_{\mathsf{upd}}(x,f,e) \stackrel{def}{=} \forall x'.\ \mathsf{alloc}_{pre}(x') \leftrightarrow \mathsf{alloc}_{post}(x') \ \wedge\ \mathsf{sel}_{post}(x,f) = e \ \wedge\ \mathsf{result} = x$$
$$\wedge\ \forall x'\ \forall f'.\ (x' \neq x \vee f' \neq f) \to \mathsf{sel}_{pre}(x',f') = \mathsf{sel}_{post}(x',f')$$

object introduction rule,

$$\text{(AL OBJ)}\quad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x_i{:}A_i{::}T_{\mathsf{res}}(x_i) \quad \forall i \in I \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j \quad \forall j \in J \end{array}}{\Gamma \vdash \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j\right]_{i\in I, j\in J} {:}A{::}T_{\mathsf{obj}}(\mathsf{f}_i{=}x_i)_{i\in I}}$$

In order to establish that the object satisfies specification A, when verifying the methods b_j we can *assume* that the self parameter y_j also satisfies A. Essentially, this causes the semantics of store specifications, to be introduced in Section 5.1 below, to be defined by a mixed-variant recursion.

The rule (AL LET) for the let case is somewhat unusual in that it introduces additional relation symbols, $\mathsf{sel}_{int}(\cdot,\cdot)$ and $\mathsf{alloc}_{int}(\cdot)$, to capture the intermediate state of the store in first-order logic. It uses *textual substitution of function* and *predicate symbols*, resp., to compose the first and second transition relation: For instance,

$$\mathsf{sel}_{post}(e_1,e_2)[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot)] \quad\equiv\quad \mathsf{sel}_{int}(e_1,e_2)$$

The side condition $[\Gamma] \vdash T$ in (AL LET) ensures that the transition relation in the conclusion respects the local scope of this intermediate state.

The remaining rules are less surprising. Rules (AL VAR) and (AL CONST) extend the corresponding type rules (TERM VAR) and (TERM CONST) of Table 3.2 on page 43 by adding the transition relation T_{res} that simply states the absence of side-effects. In the precondition of rule (AL COND), the result specification A and transition relation T may be specialised in the then and else branches. The rules (AL SEL) and (AL UPD) are also obvious extensions of the corresponding type rules, stating absence of side-effects in the former and describing the state change using T_{upd} in the latter rule. The rule (AL INV) performs the binding of the formal self parameter in A and T to the actual host object x on the logical

Table 4.4 *Inference rules of Abadi-Leino logic*

(AL SUB)
$$\frac{[\Gamma] \vdash A <: A' \quad \Gamma \vdash a{:}A{::}T \quad [\Gamma] \vdash A' \quad [\Gamma] \vdash T' \quad \vdash_{\mathsf{fo}} T \to T'}{\Gamma \vdash a{:}A'{::}T'}$$

(AL VAR)
$$\frac{\Gamma \vdash \mathsf{ok} \quad x{:}A \text{ in } \Gamma}{\Gamma \vdash x{:}A{::}T_{\mathsf{res}}(x)}$$

(AL CONST)
$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{false}{:}\mathsf{bool}{::}T_{\mathsf{res}}(\mathsf{false})} \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{true}{:}\mathsf{bool}{::}T_{\mathsf{res}}(\mathsf{true})}$$

(AL COND)
$$\frac{\begin{array}{c} A[\mathsf{true}/x] \equiv A_t[\mathsf{true}/x] \text{ and } A[\mathsf{false}/x] \equiv A_f[\mathsf{false}/x] \\ T[\mathsf{true}/x] \equiv T_t[\mathsf{true}/x] \text{ and } T[\mathsf{false}/x] \equiv T_f[\mathsf{false}/x] \\ \Gamma \vdash x{:}\mathsf{bool}{::}T_{\mathsf{res}}(x) \quad \Gamma \vdash a{:}A_t{::}T_t \quad \Gamma \vdash b{:}A_f{::}T_f \end{array}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b{:}A{::}T}$$

(AL LET)
$$\frac{\begin{array}{c} \Gamma \vdash a{:}A'{::}T' \qquad \Gamma, x{:}A' \vdash b{:}B{::}T'' \qquad [\Gamma] \vdash B \qquad [\Gamma] \vdash T \\ \vdash_{\mathsf{fo}} T'[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{post}(\cdot), x/\mathsf{result}] \\ \wedge\ T''[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{pre}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{pre}(\cdot)] \ \to\ T \end{array}}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b{:}B{::}T}$$

(AL OBJ)
$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x_i{:}A_i{::}T_{\mathsf{res}}(x_i) \quad \forall i \in I \qquad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j \quad \forall j \in J \end{array}}{\Gamma \vdash [\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J}{:}A{::}T_{\mathsf{obj}}(\mathsf{f}_i{=}x_i)_{i\in I}}$$

(AL SEL)
$$\frac{\Gamma \vdash x{:}[\mathsf{f}{:}A]{::}T_{\mathsf{res}}(x)}{\Gamma \vdash x.\mathsf{f}{:}A{::}T_{\mathsf{res}}(\mathsf{sel}_{pre}(x,\mathsf{f}))}$$

(AL UPD)
$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x{:}A{::}T_{\mathsf{res}}(x) \qquad \Gamma \vdash y{:}A_k{::}T_{\mathsf{res}}(y) \qquad k \in I \end{array}}{\Gamma \vdash x.\mathsf{f}_k := y{:}A{::}T_{\mathsf{upd}}(x,\mathsf{f}_k,y)}$$

(AL INV)
$$\frac{\Gamma \vdash x{:}[\mathsf{m}{:}\varsigma(y)A{::}T]{::}T_{\mathsf{res}}(x)}{\Gamma \vdash x.\mathsf{m}{:}A[x/y]{::}T[x/y]}$$

(AL COPY)
$$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x{:}A{::}T_{\mathsf{res}}(x) \end{array}}{\Gamma \vdash \mathsf{clone}\,x{:}A{::}\exists\overline{y}.\ \textstyle\bigwedge_{i\in I} y_i = \mathsf{sel}_{pre}(x,\mathsf{f}_i)\ \wedge\ T_{\mathsf{obj}}(\mathsf{f}_i = y_i)_{i\in I}}$$

level. The cloning rule (AL COPY) states that the result satisfies the same object specification as the cloned object; the transition relation describes the state change caused by the allocation of the clone. Finally, note that the subsumption rule incorporates a weakening of the transition relation from T to T' in addition to replacing A by a super specification A'.

4.3.3 Proving the Factorial in Abadi-Leino Logic

As a demonstration of proofs in Abadi-Leino logic, the example of the factorial program (4.2) on page 64 is considered again, this time in more detail. Thus let

$$a \equiv \left[\begin{array}{l} \mathsf{arg} = 0, \\ \mathsf{fac} = \varsigma(y).\,\mathsf{if}\ y.\mathsf{arg} = 0\ \mathsf{then}\ 1 \\ \qquad\qquad \mathsf{else\ let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x-1; x \times (y.\mathsf{fac}) \end{array}\right]$$

$$A \equiv [\mathsf{arg} : \mathsf{int},\ \mathsf{fac} : \varsigma(y)\mathsf{int}{::}T]$$

where $T \equiv \mathsf{sel}_{pre}(y,\mathsf{arg}) \geq 0 \rightarrow \mathsf{result} = \mathsf{sel}_{pre}(y,\mathsf{arg})!$. Besides the rules in Table 4.4 we assume appropriate rules for reasoning about integer expressions. We proceed bottom-up: An application of the rule (AL OBJ),

$$\frac{\vdash 0 : \mathsf{int}::T_{\mathsf{res}}(0) \qquad y{:}A \vdash \mathsf{if}\ y.\mathsf{arg}{=}0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}::T}{\vdash a : A::T_{\mathsf{obj}}(\mathsf{f} = 0)}$$

shows how a proof of $\vdash a : A :: T_{\mathsf{obj}}(\mathsf{arg} = 0)$ reduces to two proof obligations of which the first one can be immediately derived, by rule (AL CONST). In order to simplify the second one, corresponding to the method body of fac, we note that for

$$T' \equiv (e = (\mathsf{sel}_{pre}(y,\mathsf{arg}){=}0)) \rightarrow T \ \wedge\ (e \neq (\mathsf{sel}_{pre}(y,\mathsf{arg}){=}0)) \rightarrow T$$

the equivalence $\vdash_{\mathsf{fo}} T' \leftrightarrow T$ implies, by an application of (AL SUB), that proving

$$y{:}A \vdash \mathsf{if}\ y.\mathsf{arg}{=}0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}::T'$$

is sufficient. So rule (AL COND) can be applied to obtain

$$y{:}A \vdash 1 : \mathsf{int}::T'[\mathsf{true}/e] \tag{4.5}$$

$$y{:}A \vdash \mathsf{let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}::T'[\mathsf{false}/e] \tag{4.6}$$

Validity of judgement (4.5) follows by rules (AL CONST) and (AL SUB) by observing

$$\vdash_{\mathsf{fo}} T_{\mathsf{res}}(1) \rightarrow T'[\mathsf{true}/e]$$

(which in fact depends on data type axioms such as $(e{=}0) = \mathsf{true} \leftrightarrow e = 0$). Next we observe

$$\vdash_{\mathsf{fo}} (\mathsf{sel}_{pre}(y,\mathsf{arg}){>}0 \rightarrow \mathsf{result} = \mathsf{sel}_{pre}(y,\mathsf{arg})!) \quad \rightarrow \quad T'[\mathsf{false}/e]$$

By (AL SUB) it is therefore enough to prove (4.6) with $T'[\mathsf{false}/e]$ replaced by the transition relation $T'' \equiv \mathsf{sel}_{pre}(y,\mathsf{arg}){>}0 \rightarrow \mathsf{result} = \mathsf{sel}_{pre}(y,\mathsf{arg})!$. By rule (AL LET) this splits into

$$\frac{y{:}A \vdash y.\mathsf{arg} : \mathsf{int}::T_1 \qquad y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}::T_2}{y{:}A \vdash \mathsf{let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}::T''} \tag{4.7}$$

where

$$T_1 \equiv T_{\mathsf{res}}(\mathsf{sel}_{pre}(y,\mathsf{arg}))$$

$$T_2 \equiv 0 < x \ \wedge\ x = \mathsf{sel}_{pre}(y,\mathsf{arg}) \rightarrow \mathsf{result} = x \cdot (\mathsf{sel}_{pre}(y,\mathsf{arg}){-}1)!$$

so that the implication

$$T_1[\mathsf{sel}_{int}/\mathsf{sel}_{post}, \mathsf{alloc}_{int}/\mathsf{alloc}_{post}, x/\mathsf{result}] \wedge T_2[\mathsf{sel}_{int}/\mathsf{sel}_{pre}, \mathsf{alloc}_{int}/\mathsf{alloc}_{pre}] \rightarrow T''$$

is valid. The first premiss of (4.7) follows directly from (AL VAR) and (AL SEL):

$$\frac{\overline{y{:}A \vdash y : A::T_{\mathsf{res}}(y)}}{y{:}A \vdash y.\mathsf{arg} : \mathsf{int}::T_{\mathsf{res}}(\mathsf{sel}_{pre}(y,\mathsf{arg}))}$$

The second premiss of (4.7) follows by desugaring the sequential composition and applying rule (AL LET)

$$\frac{y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{arg} := x{-}1 : [\,]{::}T_2' \qquad y{:}A, x{:}\mathsf{int} \vdash x \times (y.\mathsf{fac}) : \mathsf{int}{::}T_2''}{y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{arg}{:=}x{-}1; x{\times}(y.\mathsf{fac}) : \mathsf{int}{::}T_2} \tag{4.8}$$

where the transition relations T_2' and T_2'' are

$$\begin{aligned} T_2' &\equiv \mathsf{sel}_{pre}(y, \mathsf{arg}) = x \;\rightarrow\; \mathsf{sel}_{post}(y, \mathsf{arg}) = x - 1 \\ T_2'' &\equiv 0 \le \mathsf{sel}_{pre}(y, \mathsf{arg}) \;\wedge\; \mathsf{sel}_{pre}(y, \mathsf{arg}) = x - 1 \;\rightarrow\; \mathsf{result} = x \cdot (\mathsf{sel}_{pre}(y, \mathsf{arg}))! \end{aligned}$$

Combined they imply T_2:

$$\begin{aligned} &(\mathsf{sel}_{pre}(y, \mathsf{arg}) = x \;\rightarrow\; \mathsf{sel}_{int}(y, \mathsf{arg}) = x - 1) \;\wedge \\ &(0 \le \mathsf{sel}_{int}(y, \mathsf{arg}) \;\wedge\; \mathsf{sel}_{int}(y, \mathsf{arg}) = x - 1 \;\rightarrow\; \mathsf{result} = x \cdot (\mathsf{sel}_{int}(y, \mathsf{arg}))!) \\ \rightarrow\; &(0 < x \;\wedge\; x = \mathsf{sel}_{pre}(y, \mathsf{arg}) \;\rightarrow\; \mathsf{result} = x \cdot (\mathsf{sel}_{pre}(y, \mathsf{arg}){-}1)!) \end{aligned}$$

Finally, the first premiss of (4.8) follows from (AL UPD) and a weakening step with rule (AL SUB)

$$\dfrac{\dfrac{\dfrac{}{y{:}A, x{:}\mathsf{int} \vdash y : A{::}T_{\mathsf{res}}(y)} \qquad \dfrac{}{y{:}A, x{:}\mathsf{int} \vdash x - 1 : \mathsf{int}{::}T_{\mathsf{res}}(x{-}1)}}{y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{arg} := x{-}1 : [\,]{::}T_{\mathsf{upd}}(y, \mathsf{arg}, x{-}1)}}{y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{arg} := x{-}1 : [\,]{::}T_2'}$$

The second one is more interesting, because of the recursive call. It is, however, not difficult to prove since we can use the assumption $y : A$ to obtain the necessary properties: First note that by desugaring the expression $x{\times}(y.\mathsf{fac})$,

$$\dfrac{\dfrac{}{y{:}A, x{:}\mathsf{int} \vdash x : \mathsf{int}{::}T_{\mathsf{res}}(x)} \qquad \dfrac{\dfrac{}{y{:}A, x{:}\mathsf{int} \vdash y : A{::}T_{\mathsf{res}}(y)}}{y{:}A, x{:}\mathsf{int} \vdash y.\mathsf{fac} : \mathsf{int}{::}T}}{y{:}A, x{:}\mathsf{int} \vdash x{\times}(y.\mathsf{fac}) : \mathsf{int}{::}\mathsf{sel}_{pre}(y, \mathsf{arg}) \ge 0 \rightarrow \mathsf{result} = x \cdot (\mathsf{sel}_{pre}(y, \mathsf{arg}))!}$$

from (AL VAR) and (AL INV). Hence, a further weakening step by (AL SUB) shows $y{:}A, x{:}\mathsf{int} \vdash x{\times}(y.\mathsf{fac}) : \mathsf{int}{::}T_2''$ which concludes the derivation of $a : A{::}T_{\mathsf{obj}}(\mathsf{arg} = 0)$.

This example makes clear that proof derivations by hand are rather tedious. Machine support is highly desirable, and the work of Tang and Hofmann (2002) takes an important step in this direction for Abadi and Leino's logic. Some further elaborated examples can be found in (Abadi and Leino 2004; Tang and Hofmann 2002).

4.3.4 Semantics of Specifications

Having recalled Abadi and Leino's logic, we next give a denotational semantics of specifications. Informally, a transition relation T will denote a predicate relating an initial store with a result value and the resulting store of a computation. In transition relations it is possible to quantify over field names (for an example of this see the transition relations in Table 4.3), and we write $\mathsf{Env}^* \stackrel{def}{=} \mathit{Var} \rightarrow_{\mathit{fin}} (\mathsf{Val} + \mathcal{F})$ when interpreting transition relations:

$$[\![\overline{x} \vdash T]\!] : \mathsf{Env}^* \rightarrow \mathcal{P}(\mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}})$$

Table 4.5 *Semantics of expressions*

$[\![\overline{x} \vdash e]\!] : \mathsf{Env}^* \to \mathsf{St}_{\mathsf{Val}} \to \mathsf{Val} \to \mathsf{St}_{\mathsf{Val}} \rightharpoonup (\mathsf{Val} + \mathcal{F})$

$$[\![\overline{x} \vdash x]\!]\,\rho\sigma\nu\sigma' = \begin{cases} \rho(x) & \text{if } x \in \mathsf{dom}(\rho) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\overline{x} \vdash \mathsf{f}]\!]\,\rho\sigma\nu\sigma' = \mathsf{f}$$

$$[\![\overline{x} \vdash \mathsf{result}]\!]\,\rho\sigma\nu\sigma' = \nu$$

$$[\![\overline{x} \vdash \mathsf{true}]\!]\,\rho\sigma\nu\sigma' = \mathit{true}$$

$$[\![\overline{x} \vdash \mathsf{false}]\!]\,\rho\sigma\nu\sigma' = \mathit{false}$$

$$[\![\overline{x} \vdash \mathsf{sel}_{\mathit{pre}}(e_0, e_1)]\!]\,\rho\sigma\nu\sigma' = \begin{cases} \sigma.l.\mathsf{f} & \text{if } [\![\overline{x} \vdash e_0]\!]\,\rho\sigma\nu\sigma' = l \in \mathsf{Loc} \text{ defined} \\ & \text{and } [\![\overline{x} \vdash e_1]\!]\,\rho\sigma\nu\sigma' = \mathsf{f} \in \mathcal{F} \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\overline{x} \vdash \mathsf{sel}_{\mathit{post}}(e_0, e_1)]\!]\,\rho\sigma\nu\sigma' = \begin{cases} \sigma'.l.\mathsf{f} & \text{if } [\![\overline{x} \vdash e_0]\!]\,\rho\sigma\nu\sigma' = l \in \mathsf{Loc} \text{ defined} \\ & \text{and } [\![\overline{x} \vdash e_1]\!]\,\rho\sigma\nu\sigma' = \mathsf{f} \in \mathcal{F} \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Table 4.6 *Semantics of transition relations*

$[\![\overline{x} \vdash T]\!] : \mathsf{Env}^* \to \mathcal{P}(\mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}})$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash e_0 = e_1]\!]\,\rho \overset{\mathit{def}}{\iff} \begin{cases} \text{both } [\![\overline{x} \vdash e_0]\!]\,\rho\sigma\nu\sigma' \text{ and } [\![\overline{x} \vdash e_1]\!]\,\rho\sigma\nu\sigma' \text{ are defined} \\ \text{and equal, or both undefined} \end{cases}$$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash \mathsf{alloc}_{\mathit{pre}}(e)]\!]\,\rho \overset{\mathit{def}}{\iff} [\![\overline{x} \vdash e]\!]\,\rho\sigma\nu\sigma' \downarrow \ \wedge\ [\![\overline{x} \vdash e]\!]\,\rho\sigma\nu\sigma' \in \mathsf{dom}(\sigma)$$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash \mathsf{alloc}_{\mathit{post}}(e)]\!]\,\rho \overset{\mathit{def}}{\iff} [\![\overline{x} \vdash e]\!]\,\rho\sigma\nu\sigma' \downarrow \ \wedge\ [\![\overline{x} \vdash e]\!]\,\rho\sigma\nu\sigma' \in \mathsf{dom}(\sigma')$$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash \neg T]\!]\,\rho \overset{\mathit{def}}{\iff} (\sigma, \nu, \sigma') \notin [\![\overline{x} \vdash T]\!]\,\rho$$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash T_0 \wedge T_1]\!]\,\rho \overset{\mathit{def}}{\iff} (\sigma, \nu, \sigma') \in [\![\overline{x} \vdash T_0]\!]\,\rho \cap [\![\overline{x} \vdash T_1]\!]\,\rho$$

$$(\sigma, \nu, \sigma') \in [\![\overline{x} \vdash \forall x.T]\!]\,\rho \overset{\mathit{def}}{\iff} \text{for all } u \in \mathsf{Val} \uplus \mathcal{F}.\ (\sigma, \nu, \sigma') \in [\![\overline{x}, x \vdash T]\!]\,\rho[x := u]$$

The ability of variables being bound to values as well as field names is related to the distinction between program variables and auxiliary variables in Hoare logics, although transition relations are used here in place of Hoare triples. In particular, allowing quantification over locations *and* field names is necessary to express invariants and relate pre- and post-execution stores, as exemplified by the relations in Table 4.3 (but see Remark 4.3.2 below).

The definition of the semantics of expressions, $[\![\overline{x} \vdash e]\!]$, requires some care since it may be undefined, see Table 4.5. Once the semantics of expressions is settled, the semantics of transition relations can be defined in a straightforward way; a few typical cases are given in Table 4.6.

Note that even though expressions may be undefined (for example, because of referring to non-existent fields), the interpretation of transition relations is two-valued. Also observe that the meaning of a transition relation $\overline{x} \vdash T$ without free variables does not

Table 4.7 *Semantics of specifications*

$$[\![\overline{x} \vdash A]\!] : \mathsf{Env} \to \mathcal{P}(\mathsf{Val} \times \mathsf{St})$$

$$[\![\overline{x} \vdash \mathsf{bool}]\!]\,\rho \stackrel{def}{=} \mathsf{BVal} \times \mathsf{St}$$

$$[\![\overline{x} \vdash [\mathsf{f}_i\colon A_i,\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}]\!]\,\rho$$

$$\stackrel{def}{=} \left\{ \langle l, \sigma\rangle \in \mathsf{Loc}\times\mathsf{St} \;\middle|\; \begin{array}{ll} \text{(i)} & \forall i \in I.\ \langle \sigma.l.\mathsf{f}_i, \sigma\rangle \in [\![\overline{x} \vdash A_i]\!]\,\rho \\ \text{(ii)} & \forall j \in J.\ \text{if } \sigma.l.\mathsf{m}_j(l,\sigma) = \langle v, \sigma'\rangle \\ & \quad \text{then } \langle v, \sigma'\rangle \in [\![\overline{x}, y_j \vdash B_j]\!]\,\rho[y_j := l] \\ & \quad \text{and } (\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x}, y_j \vdash T_j]\!]\,\rho[y_j := l] \end{array} \right\}$$

depend on the environment. Therefore we may omit the environment and simply write $[\![T]\!]$ for closed T.

Remark 4.3.2. *We think it would be clearer to use a multi-sorted logic, with different quantifiers ranging over locations, basic values and field names, resp., but decided to keep close to the original presentation of Abadi and Leino's logic.*

An object specification $\overline{x} \vdash A$ gives rise to a predicate that depends on values for the free variables. At first glance, this is reminiscent of dependent types, such as system λP (for instance, see Barendregt 1992).

However, since the underlying logic in the transition relations is untyped, the types of the free variables are not relevant.The interpretation of object specifications $\overline{x} \vdash A$,

$$[\![\overline{x} \vdash A]\!] : \mathsf{Env} \to \mathcal{P}(\mathsf{Val} \times \mathsf{St})$$

is given in Table 4.7.

We begin with a simple observation about the interpretation which will be used to obtain soundness of the proof system in the sense of "well-typed (or rather, *verified*) programs do not raise error".

Lemma 4.3.3. *For all specifications $\overline{x} \vdash A$, stores $\sigma \in \mathsf{St}$ and environments $\rho \in \mathsf{Env}$,*

$$\langle \mathsf{error}, \sigma\rangle \notin [\![\overline{x} \vdash A]\!]\,\rho$$

Proof. Immediate from the definition of $[\![\overline{x} \vdash A]\!]\,\rho$. □

The following results establish that the denotation of a specification $\overline{x} \vdash A$ is downwards closed and admissible.

Lemma 4.3.4 (Downward Closure)**.** *For all specifications $\overline{x} \vdash A$, stores $\sigma, \sigma' \in \mathsf{St}$, values $v \in \mathsf{Val}$ and environments $\rho \in \mathsf{Env}$,*

$$\sigma' \sqsubseteq \sigma \ \wedge\ \langle v, \sigma\rangle \in [\![\overline{x} \vdash A]\!]\,\rho \quad \Longrightarrow \quad \langle v, \sigma'\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$$

Proof. The proof proceeds by induction on A. In the case where A is bool this is immediate. So suppose A is a specification of the form $\overline{x} \vdash [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ and let $\langle l, \sigma\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$.

By definition of $\sigma' \sqsubseteq \sigma$ we have $\sigma'.l.\mathsf{f}_i = \sigma.l.\mathsf{f}_i$ for all $i \in I$, which by induction hypothesis entails $\langle \sigma'.l.\mathsf{f}_i, \sigma'\rangle \in [\![\overline{x} \vdash A_i]\!]\,\rho$. Next, $\sigma'.l.\mathsf{m}_j \sqsubseteq \sigma.l.\mathsf{m}_j$ for all $j \in J$, again from $\sigma' \sqsubseteq \sigma$. Thus, if $\sigma'.l.\mathsf{m}_j(l, \sigma')\downarrow$ then monotonicity of $\sigma'.l.\mathsf{m}_j$ and the assumption $\langle l, \sigma\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$ yield $\sigma.l.\mathsf{m}_j(l, \sigma) = \langle v, \sigma_2\rangle$ for some $\langle v, \sigma_2\rangle \in [\![\overline{x}, y_j \vdash B_j]\!]\,\rho[y_j := l]$ such that $\langle v, \sigma_1\rangle \sqsubseteq \langle v, \sigma_2\rangle$ and $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma_2)) \in [\![\overline{x}, y_j \vdash T_j]\!]\,\rho[y_j := l]$, where $\sigma'.l.\mathsf{m}_j(l, \sigma') = \langle v, \sigma_1\rangle$. Induction yields $\langle v, \sigma_1\rangle \in [\![\overline{x}, y_j \vdash B_j]\!]\,\rho[y_j := l]$, and the condition $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma_1)) \in [\![\overline{x}, y_j \vdash T_j]\!]\,\rho[y_j := l]$ follows since $\pi_{\mathrm{Val}}(\sigma') = \pi_{\mathrm{Val}}(\sigma)$ and $\pi_{\mathrm{Val}}(\sigma_1) = \pi_{\mathrm{Val}}(\sigma_2)$. Hence we have shown that $\langle l, \sigma'\rangle \in [\![\overline{x} \vdash A_i]\!]\,\rho$ as required. □

Lemma 4.3.5 (Admissibility)**.** *For all specifications $\overline{x} \vdash A$, stores $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ in* St*, values $v \in \mathsf{Val}$ and environments $\rho \in \mathsf{Env}$,*

$$\forall n \in \mathbb{N}.\ \langle v, \sigma_n\rangle \in [\![\overline{x} \vdash A]\!]\,\rho \quad \Longrightarrow \quad \langle v, \sigma\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$$

where $\sigma \stackrel{def}{=} \bigsqcup_n \sigma_n$.

Proof. The proof is by an induction on A. In the case where A is bool there is nothing to prove since $[\![\overline{x} \vdash \mathsf{bool}]\!]$ does not depend on the store. Now assume that $\overline{x} \vdash A$ is of the form $\overline{x} \vdash [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ and let $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ be a countable chain such that $\langle l, \sigma_n\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$ for all $n \in \mathbb{N}$.

Let $\sigma = \bigsqcup_n \sigma_n$, then from $\sigma.l.\mathsf{f}_i = \sigma_n.l.\mathsf{f}_i$ for all n and induction hypothesis we obtain $\langle \sigma.l.\mathsf{f}_i, \sigma\rangle \in [\![\overline{x} \vdash A_i]\!]\,\rho$, for all $i \in I$. Next, suppose $\sigma.l.\mathsf{m}_j(l, \sigma)\downarrow$, i.e., $\sigma.l.\mathsf{m}_j(l, \sigma) = \langle v, \sigma'\rangle$ for some $v \in \mathsf{Val}$ and $\sigma' \in \mathsf{St}$. By continuity, $\sigma_n.l.\mathsf{m}_j(l, \sigma_n)\downarrow$ for all sufficiently large $n \in \mathbb{N}$, i.e., $\sigma_n.l.\mathsf{m}_j(l, \sigma_n) = \langle v, \sigma'_n\rangle$. By assumption, $\langle v, \sigma'_n\rangle \sqsubseteq \langle v, \sigma'_{n+1}\rangle \sqsubseteq \ldots$ is a countable chain in $[\![\overline{x}, y_j \vdash B_j]\!]\,\rho[y_j := l]$ and $(\pi_{\mathrm{Val}}(\sigma_n), v, \pi_{\mathrm{Val}}(\sigma'_n)) \in [\![\overline{x}, y_j \vdash T_j]\!]\,\rho[y_j := l]$ for all sufficiently large n. By induction, the former yields $\langle v, \sigma'\rangle \in [\![\overline{x}, y_j \vdash B_j]\!]\,\rho[y_j := l]$. Moreover, since $\pi_{\mathrm{Val}}(\sigma) = \pi_{\mathrm{Val}}(\sigma_n) = \pi_{\mathrm{Val}}(\sigma_{n+1}) = \ldots$ and $\pi_{\mathrm{Val}}(\sigma') = \pi_{\mathrm{Val}}(\sigma'_n) = \pi_{\mathrm{Val}}(\sigma'_{n+1}) = \ldots$ we also find $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x}, y_j \vdash T_j]\!]\,\rho[y_j := l]$. This concludes the proof that indeed $\langle l, \sigma\rangle \in [\![\overline{x} \vdash A]\!]\,\rho$. □

Lemma 4.3.6 (Subspecification)**.** *Suppose $\overline{x} \vdash A \prec: B$. Then, for all environments $\rho \in \mathsf{Env}$,*

$$[\![\overline{x} \vdash A]\!]\,\rho \subseteq [\![\overline{x} \vdash B]\!]\,\rho$$

Proof. This follows by induction on the derivation of $\overline{x} \vdash A \prec: B$. The cases for reflexivity and transitivity are immediate. For the case where $\overline{x} \vdash A \prec: B$ has been derived by (SUBSPEC OBJ), both A and B are object specifications, and we need a similar lemma for transition relations:

If $\overline{x} \vdash T$ and $\overline{x} \vdash T'$ then

$$\vdash_{\mathsf{fo}} T \to T' \quad \Longrightarrow \quad [\![\overline{x} \vdash T]\!]\,\rho \ \subseteq\ [\![\overline{x} \vdash T']\!]\,\rho \tag{4.9}$$

for all $\rho \in \mathsf{Env}^*$. However, (4.9) follows immediately since $\vdash_{\mathsf{fo}}$ holds in all models of first-order logic. The result then follows from the induction hypothesis and (4.9) by definition of the semantics (Table 4.7). □

Chapter 5

Soundness of Abadi and Leino's Logic

In this chapter we prove soundness of Abadi and Leino's logic with respect to the denotational semantics. In Section 5.1 we argue the necessity of introducing the concept of *store specifications*. After illustrating that the "obvious" definition of their semantics cannot be established using standard techniques (Section 5.1.2), we use the domain-theoretic techniques from Chapter 2 to establish the well-definedness of an alternative semantics in Section 5.1.3. Section 5.2 finally presents the soundness proof. After establishing some preliminary results, the main lemma is proved in Section 5.2.2; soundness of the logic follows easily from this.

5.1 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 in both operational and denotational settings (for instance, see (Abadi and Cardelli 1996, Chap. 11) and the references therein, and (Levy 2002), respectively). Since statements may call subprograms residing in the store, the store has to be checked as well. However, the store may contain loops and therefore induction on the reachable part of the store is unavailable.

The standard remedy – also used in (Abadi and Leino 2004) – is to relativise the typing judgement (here: "specification judgement") such that it only needs to hold for well-typed (here: "verified") stores. In other words, judgements are interpreted with respect to *store specifications*. A store specification Σ assigns a specification to each location in a store:

$$\Sigma \quad \equiv \quad l_1{:}A_1, \ldots, l_n{:}A_n$$

When an object is created, the specification assigned to it at the time of creation is included in the store specification, leading to *store specification extensions*. In this sense, store specifications are a dynamic concept.

In this section we will interpret such store specifications using the techniques from previous work of Reus and Streicher (2004). Since the denotation of a store specification will rely on mixed-variant recursion, we were not able to define a semantic notion of

subspecification for stores. However, the logic of Abadi and Leino makes essential use of subspecifications.

We get around this problem by only using a subset relationship on denotations of object specifications. In object specifications there is no contravariant occurrence of store, as the semantics of objects is with respect to one fixed store (see Table 4.7).

We are restricted by the logic's requirement that verified statements never break the validity of store specifications. Suppose y denotes an object residing in a store σ that satisfies a specification B. For a field update $\sigma.l.\mathsf{f} := y$ to preserve a specification $l : A$ where $A \prec: [\mathsf{f} : B]$, the location l must be in the set

$$\{l \in \mathsf{Loc} \mid \Sigma.l \equiv A' \quad \wedge \quad \vdash A' \prec: A\} \tag{5.1}$$

which ensures that A and A' have equal f components B. Since the semantic interpretation of the subspecification relation as set containment cannot reflect this invariance, preservation can *not* be guaranteed for locations in the (semantically more appealingly defined) set

$$\{l \in \mathsf{dom}(\sigma) \mid \langle l, \sigma\rangle \in [\![A]\!]\} \tag{5.2}$$

Therefore we were forced to use the former set (5.1) for the interpretation of A in the semantics of store specifications. Unfortunately this means we still rely on the (synactically defined) notion of subspecification.

5.1.1 Result Specifications, Store Specifications and a Tentative Semantics

A store specification Σ assigns *closed* specifications $\vdash A$ to (a finite set of) locations:

Definition 5.1.1 (Store Specifications)**.** *A record $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Spec)$ is a* store specification *if for all $l \in dom(\Sigma)$, $\Sigma.l = A$ is a closed object specification. Let StSpec denote the set of store specifications.*

Because we focus on closed specifications in the following, we need a way to turn the components B_j of a specification $[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ (which in general will depend on the self parameter y_j) into closed specifications. We do this by extending the syntax of expressions with locations: There is one symbol $\underline{l}$ for each $l \in \mathsf{Loc}$, and define $[\![\overline{x} \vdash \underline{l}]\!]\,\rho \stackrel{def}{=} l$ (see Table 4.6). Similarly, we set $\underline{true} \stackrel{def}{=} \mathsf{true}$ and $\underline{false} \stackrel{def}{=} \mathsf{false}$ for the elements of BVal. When clear from context we will simply write v in place of $\underline{v}$.

Further we write $A[\rho/\overline{x}]$ (and $A[\rho/\Gamma]$, respectively) for the simultaneous substitution of all $x \in \overline{x}$ ($x \in [\Gamma]$, respectively) in A by $\underline{\rho(x)}$. Then we can prove the following substitution lemma.

Lemma 5.1.2 (Substitution)**.** *Let $\rho \in \mathsf{Env}^*$ be an environment such that $\overline{x} \subseteq dom(\rho)$. Suppose $\overline{x} \vdash T$ is a transition relation and $\overline{x} \vdash A$ and $\overline{x} \vdash A'$ are specifications. Then the following hold:*

- $\vdash T[\rho/\overline{x}]$ *and* $[\![\vdash T[\rho/\overline{x}]]\!] = [\![\overline{x} \vdash T]\!]\,\rho$
- $\vdash A[\rho/\overline{x}]$ *and* $[\![\vdash A[\rho/\overline{x}]]\!] = [\![\overline{x} \vdash A]\!]\,\rho$

- *if* $\overline{x} \vdash A \prec: A'$ *then* $\vdash A[\rho/\overline{x}] \prec: A'[\rho/\overline{x}]$

Proof. The first part is by induction on T, the second by induction on A and the last by induction on the derivation of $\overline{x} \vdash A \prec: A'$. □

Definition 5.1.3 (Store Specification Extension)**.** *Let* $\Sigma, \Sigma' \in \mathit{StSpec}$ *be store specifications.* Σ' extends Σ*, written*

$$\Sigma' \succcurlyeq \Sigma,$$

if $\mathit{dom}(\Sigma) \subseteq \mathit{dom}(\Sigma')$ *and* $\Sigma.l \equiv \Sigma'.l$ *for all* $l \in \mathit{dom}(\Sigma)$*.*

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 5.1.4 (Object Specifications)**.** *Suppose* $\Sigma \in \mathit{StSpec}$ *is a store specification. For closed* $\vdash A$ *let* $\|A\|_\Sigma \subseteq \mathsf{Val}$ *be defined by*

$$\begin{aligned} \|\mathsf{bool}\|_\Sigma &\stackrel{def}{=} \mathsf{BVal} \\ \|B\|_\Sigma &\stackrel{def}{=} \{l \in \mathsf{Loc} \mid \vdash \Sigma.l \prec: B\} \end{aligned}$$

where $B \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ *is an object specification. We extend this definition to specification contexts* $\|\Gamma\|_\Sigma \subseteq \mathsf{Env}$ *in the natural way:*

$$\begin{aligned} \rho \in \|\emptyset\|_\Sigma &\stackrel{def}{\iff} \mathit{always} \\ \rho \in \|\Gamma, x{:}A\|_\Sigma &\stackrel{def}{\iff} \rho \in \|\Gamma\|_\Sigma \quad\wedge\quad \rho(x) \in \|A[\rho/\Gamma]\|_\Sigma \end{aligned}$$

Remark 5.1.5. *Note that StSpec is partially ordered by* $\succcurlyeq$*. Further, observe that for all* A*,*

$$\Sigma' \succcurlyeq \Sigma \quad\Longrightarrow\quad \|A\|_\Sigma \subseteq \|A\|_{\Sigma'}$$

Thus, $\|A\| : \mathit{StSpec} \longrightarrow \mathcal{A}dm(\mathsf{Val})$ *is a functor from the partial order category StSpec to the category of (trivially admissible) subsets of* Val *ordered by set inclusion. This observation provides a link to the possible worlds model of types due to Levy (2004), which is used in Chapters 8 and 9.*

We obtain the following lemma about *context extensions*.

Lemma 5.1.6 (Context Extension)**.** *If* $\rho \in \|\Gamma\|_\Sigma$ *and* $\Gamma, x{:}A \vdash \mathsf{ok}$ *and* $v \in \|A[\rho/\Gamma\|_\Sigma$ *then*

$$\rho[x := v] \in \|\Gamma, x{:}A\|_\Sigma$$

Proof. The result follows immediately from the definition once we show $\rho[x := v] \in \|\Gamma\|_\Sigma$. This can be seen to hold since $x \notin \mathsf{dom}(\Gamma)$ by $\Gamma, x{:}A \vdash \mathsf{ok}$, hence for all $y{:}B$ in Γ we know that $x \notin \mathsf{fv}(B)$ and we must have $B[\rho[x := v]/\Gamma] \equiv B[\rho/\Gamma]$. □

In light of the object introduction rule (AL Obj), we would like to interpret store specifications as predicates over stores, as follows. Let $\mathcal{R} \stackrel{def}{=} \mathcal{P}(\mathsf{St})^{\mathit{StSpec}}$ denote the collection of predicates on St, indexed by store specifications, and define a functional $\Phi : \mathcal{R}^{op} \times \mathcal{R} \to \mathcal{R}$

Table 5.1 *Store specifications, first (and incorrect) attempt*

$$
\begin{array}{lll}
\sigma \in \Phi(Y,X)_\Sigma & \stackrel{def}{\iff} & \forall l \in \mathsf{dom}(\Sigma) \text{ where } \Sigma.l \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}: \\
& & \text{(F)} \quad \forall i \in I.\ \ \sigma.l.\mathsf{f}_i \in \|A_i\|_\Sigma; \quad \text{and} \\
& & \text{(M)} \quad \forall j \in J.\ \ \forall \Sigma' \succcurlyeq \Sigma\ \forall \sigma', \sigma'' \in \mathsf{St}\ \forall l' \in \mathsf{Loc}\ \forall v \in \mathsf{Val}. \\
& & \qquad l' \in \|\Sigma.l\|_{\Sigma'} \ \wedge\ \sigma' \in Y_{\Sigma'} \ \wedge\ \sigma.l.\mathsf{m}_j(l',\sigma') = \langle v, \sigma''\rangle \quad \Longrightarrow \\
& & \text{(M1)} \quad (\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l'/y_j]]\!] \qquad \text{and} \\
& & \text{(M2)} \quad \exists \Sigma'' \in \mathit{StSpec}.\ \Sigma'' \succcurlyeq \Sigma' \ \wedge\ \sigma'' \in X_{\Sigma''} \quad \text{and} \\
& & \text{(M3)} \quad v \in \big|\big|B_j[l'/y_j]\big|\big|_{\Sigma''}
\end{array}
$$

by the equations in Table 5.1. We wish to set $[\![\Sigma]\!] \stackrel{def}{=} \mathit{fix}(\Phi)_\Sigma$ to obtain the semantics of Σ as a predicate on St.

The universal quantification over extensions Σ' of Σ in (M) accounts for the specifications of objects allocated between time of definition and time of call of methods. The existential quantification over extensions Σ'' of 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 for other languages with dynamic allocation, for instance (Levy 2002; Levy 2004; Reddy and Yang 2004; Stark 1998).

Unfortunately, there is a problem with this "definition" of $[\![\Sigma]\!]$ as $\mathit{fix}(\Phi)_\Sigma$, which we discuss next.

5.1.2 On the Existence of Store Specifications

The contravariant occurrence of Y in case (M) of the "definition" of Φ above is forced by the premise of the object construction rule in the Abadi-Leino logic. It states that, in order to prove that specification A holds for a new object, one can assume that the self object in methods already fulfils the specification A. As illustrated with the adequacy proof in Chapter 3, it is this contravariance, in turn, that calls for some advanced domain theory to show that the fixed point of Φ actually exists.

Unfortunately, the usual techniques (Pitts 1996; Reus and Streicher 2004) for establishing the existence of such predicates involving a mixed-variance recursion (suitably extended to *families* of predicates as summarised in Chapter 2) do not apply: They require the functional Φ of the above recursion to map *admissible predicates to admissible predicates*. Due to the existential quantification in (M2) and (M3), ranging over extensions of the store specification Σ', this property turns out to fail in this instance.

To see the impossibility of defining $[\![\Sigma]\!]$ this way, consider the following example. Let $\Sigma \in \mathit{StSpec}$ be

$$\Sigma \quad \stackrel{def}{=} \quad l_0 : [\mathsf{m}_0 : \varsigma(x)[\mathsf{m}_1 : \varsigma(y)[]{::}\mathit{True}]{::}\mathit{True}]$$

which, informally, describes a store with a single object at location l_0 containing a method

m_0. In case a call of this method converges it returns an object satisfying $[\mathsf{m}_1 : \varsigma(y)[]::\mathit{True}]$ (which is not much of a restriction). However, this result object has to be allocated in the store, and so a proper extension of the original store specification Σ has to be found.

Next, for $i \in \mathbb{N}$ let A_i be the object specification defined inductively by

$$\begin{aligned} A_0 &\stackrel{def}{=} [\mathsf{m}_1 : \varsigma(y)[]::\mathit{False}] \\ A_{i+1} &\stackrel{def}{=} [\mathsf{m}_1 : \varsigma(y)A_i::\mathit{True}] \end{aligned}$$

In particular, this means that the method m_1 of objects satisfying A_0 *must* diverge. The method m_1 of an object satisfying A_i returns an object satisfying A_{i-1}. Hence, for such objects x, it is possible to have method calls $x.\mathsf{m}_1.\mathsf{m}_1 \ldots \mathsf{m}_1$ at most i times, of which the i-th call must necessarily diverge (the others may or may not terminate).

The example below uses the fact that we can construct an ascending chain of objects for which the first $i-1$ calls indeed terminate, and therefore do *not* satisfy A_{i-1}. Then, the limit of this chain is an object x for which an arbitrary number of calls $x.\mathsf{m}_1.\mathsf{m}_1 \ldots \mathsf{m}_1$ terminates, and which therefore does not satisfy *any* of the A_i: Set $\Sigma_i'' \stackrel{def}{=} \Sigma, l : A_i$ and let $\underline{\sigma} \in [\![\Sigma]\!]$ denote some store satisfying Σ according to the tentative definition above. Moreover, define

$$\sigma_i \stackrel{def}{=} \{\!|l_0 = \{\!|\mathsf{m}_0 = \lambda\_.\langle l, \underline{\sigma} + \sigma_i''\rangle|\!\}|\!\}$$

where $\sigma_0'' \stackrel{def}{=} \{\!|l = \{\!|\mathsf{m}_1 = \lambda\_.\bot|\!\}|\!\}$ and $\sigma_{i+1}'' \stackrel{def}{=} \{\!|l = \{\!|\mathsf{m}_1 = \lambda\_.\langle l, \underline{\sigma} + \sigma_i''\rangle|\!\}|\!\}$, and let $\sigma \stackrel{def}{=} \bigsqcup_i \sigma_i$. Finally, define (indexed) relations $X, Y \in \mathcal{R}$ by

$$X_{\hat{\Sigma}} \stackrel{def}{=} \begin{cases} \{\underline{\sigma} + \sigma_i''\} & \text{if } \exists i \in \mathbb{N}.\, \hat{\Sigma} \equiv \Sigma_i'' \\ \emptyset & \text{otherwise} \end{cases}$$

$$Y_{\hat{\Sigma}} \stackrel{def}{=} \begin{cases} \{\underline{\sigma}\} & \text{if } \hat{\Sigma} \equiv \Sigma \\ \emptyset & \text{otherwise} \end{cases}$$

By construction, both X and Y are admissible in every component $\hat{\Sigma}$. By induction one obtains $\sigma_0'' \sqsubseteq \sigma_1'' \sqsubseteq \ldots$, therefore $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ in $\Phi(Y, X)_\Sigma \subseteq \mathsf{St}$. Hence we must show $\sigma \in \Phi(Y, X)_\Sigma$. But this is not the case, since it would entail, by (M2) and

$$\sigma.l_0.\mathsf{m}_0(l, \underline{\sigma}) = \bigsqcup_i \sigma_i.l_0.\mathsf{m}_0(l, \underline{\sigma}) = \langle l, \underline{\sigma} + \bigsqcup_i \sigma_i''\rangle$$

that there exists $\Sigma'' \succcurlyeq \Sigma$ such that $\underline{\sigma} + \bigsqcup_i \sigma_i'' \in X_{\Sigma''}$. Clearly this does not hold: $\underline{\sigma} + \bigsqcup_i \sigma_i''$ is *strictly* above every $\underline{\sigma} + \sigma_i''$ and therefore not in any of the $X_{\Sigma_i''}$. Hence, by choice of X, there is no store specification $\Sigma'' \succcurlyeq \Sigma$ such that $\underline{\sigma} + \sigma_i'' \in X_{\Sigma''}$. This shows that $\Phi(Y, X)_\Sigma$ is not necessarily admissible, even if X (and also Y) is.

More abstractly, this (counter-) example relies on the fact that we are dealing with *families* $X = (X_w)_{w \in \mathcal{W}}$ of predicates. Due to the existential quantification over the indices $w \in \mathcal{W}$ (store specifications $\Sigma'' \in \mathit{StSpec}$ in the case of Table 5.1), it is possible to pick *different* w_i for each element of an ω-chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ so that $f_i(x) \in X_{w_i}$. Thus, in general there need not be $w \in \mathcal{W}$ such that $f(x) \in X_w$ holds for the lub $f = \bigsqcup_i f_i$, even under the assumption that each X_w is closed under taking least upper bounds.

5.1.3 A Refined Semantics of Store Specifications

We refine the definition of store predicates by replacing the existential quantifier in (M2) of the functional Φ on page 80 by a *choice function*, as follows: We call the elements of the recursively defined domain

$$\phi \in \mathsf{RSF} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times \mathsf{St} \times \mathsf{RSF} \times \mathit{StSpec} \rightharpoonup \mathit{StSpec} \times \mathsf{RSF})) \tag{5.3}$$

(records of) *choice functions*, or *Skolem Functions*. The intuition is that, given a store $\sigma \in [\![\Sigma]\!]$ and some extension $\Sigma' \succcurlyeq \Sigma$, if $\sigma' \in [\![\Sigma']\!]$ with choice function ϕ' and the method invocation $\sigma.l.\mathsf{m}(l', \sigma')$ terminates with a store σ'', then $\phi.l.\mathsf{m}(l', \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$ yields a store specification $\Sigma'' \succcurlyeq \Sigma'$ such that $\sigma'' \in [\![\Sigma'']\!]$ and ϕ'' is a choice function for the extension Σ'' of Σ. This is again an abstraction of the actual store σ, this time abstracting the *dynamic effects* of methods with respect to allocation, on the level of store specifications. Note that the argument store σ' is needed in general to determine the resulting extension of the specification, for instance since allocation behaviour may depend on the actual values of fields.

RSF is obtained as minimal invariant of a locally continuous bifunctor $F_{\mathsf{RSF}}(\mathsf{St}, -, -)$ on **pCpo**, where

$$F_{\mathsf{RSF}}(R, S, T) \stackrel{def}{=} \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times R \times S \times \mathit{StSpec} \rightharpoonup \mathit{StSpec} \times T)) \tag{5.4}$$

We use the domain RSF of choice functions explicitly in the interpretation of store specifications below. This has the effect of constraining the existential quantifier to work *uniformly* on the elements of increasing chains, thereby precluding the counter-example to admissibility of the previous subsection.

Definition 5.1.7 (Store Predicate, Revisited)**.** *Let* $S \stackrel{def}{=} (\mathcal{A}dm(\mathsf{St} \times \mathsf{RSF}))^{\mathit{StSpec}}$ *denote the collection of families of admissible subsets of* $\mathsf{St} \times \mathsf{RSF}$*, indexed by store specifications. We define a functional* $\Phi : S^{op} \times S \to S$ *in Table 5.2. We write* $\sigma \in [\![\Sigma]\!]$ *if there is some* $\phi \in \mathsf{RSF}$ *such that* $\langle \sigma, \phi \rangle \in \mathit{fix}(\Phi)_\Sigma$*.*

Lemma 5.1.8. *Functional* Φ*, defined in Table 5.2, has a unique fixed point,* $X \stackrel{def}{=} \mathit{fix}(\Phi) \in S$*, such that* $X = \Phi(X, X)$*.*

Of course we want to use the machinery of Chapter 2 to prove Lemma 5.1.8. Thus we define a relational structure $\mathcal{R}$ next: For each pair of cpos $\langle D, E \rangle$ let

$$\mathcal{R}(D, E) \quad \stackrel{def}{=} \quad (\mathcal{A}dm(D \times E))^{\mathit{StSpec}}$$

be the chain-closed subsets of the cpo $D \times E$, indexed by store specifications $\Sigma \in \mathit{StSpec}$. Further, for $X \in \mathcal{R}(D, D')$, $Y \in \mathcal{R}(E, E')$ and partial continuous maps $e : D \rightharpoonup E$ and $e' : D' \rightharpoonup E'$, let

$$\begin{aligned} \langle e, e' \rangle : X \subset Y \quad \stackrel{def}{\iff} \quad & \forall \Sigma \in \mathit{StSpec}\ \forall \sigma \in D\ \forall \phi \in D'. \\ & \langle \sigma, \phi \rangle \in X_\Sigma \ \wedge\ (e(\sigma)\!\downarrow\ \vee\ e'(\phi)\!\downarrow) \quad \Longrightarrow \quad \langle e(\sigma), e'(\phi) \rangle \in Y_\Sigma \end{aligned}$$

Thus, in the case where $D = \mathsf{St} = E$ and $D' = \mathsf{RSF} = E'$ the statement $e : X \subset X'$ means that e maps pairs of stores and choice functions that are in X_Σ to pairs of stores and choice functions that are in the corresponding component X'_Σ of X'.

Table 5.2 *Store predicate*

$$
\begin{array}{lll}
\langle \sigma, \phi \rangle \in \Phi(Y,X)_\Sigma & \stackrel{def}{\iff} & \text{(DOM)}\quad \mathsf{dom}(\Sigma) = \mathsf{dom}(\phi) \wedge \\
& & \qquad \forall l \in \mathsf{dom}(\Sigma).\, \mathsf{dom}(\Sigma.l) = [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \implies \\
& & \qquad\quad \mathsf{dom}(\phi.l) = \{\mathsf{m}_j\}_{j\in J};\ \text{and} \\
& & \forall l \in \mathsf{dom}(\Sigma) \text{ where } \Sigma.l \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}: \\
& & \text{(F)}\quad \forall i \in I.\ \sigma.l.\mathsf{f}_i \in \|A_i\|_\Sigma;\ \text{and} \\
& & \text{(M)}\quad \forall j \in J.\ \forall \Sigma' \succcurlyeq \Sigma\ \forall \sigma', \sigma'' \in \mathsf{St}\ \forall \phi' \in \mathsf{RSF}\ \forall l' \in \mathsf{Loc}\ \forall v \in \mathsf{Val}. \\
& & \qquad l' \in \|\Sigma.l\|_{\Sigma'} \wedge \langle \sigma', \phi' \rangle \in Y_{\Sigma'} \wedge \sigma.l.\mathsf{m}_j(l', \sigma') = \langle v, \sigma'' \rangle \implies \\
& & \qquad\quad \exists \Sigma'' \succcurlyeq \Sigma'\ \exists \phi'' \in \mathsf{RSF}.\ \phi.l.\mathsf{m}_j(l', \sigma', \phi', \Sigma') = (\Sigma'', \phi'') \wedge \\
& & \text{(M1)}\quad (\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l'/y_j]]\!] \\
& & \text{(M2)}\quad \langle \sigma'', \phi'' \rangle \in X_{\Sigma''} \\
& & \text{(M3)}\quad v \in ||B_j[l'/y_j]||_{\Sigma''}
\end{array}
$$

Lemma 5.1.9. *$\mathcal{R}$ defines a normal relational structure on the bilimit-compact product category* **pCpo** $\times$ **pCpo**. *Inverse images are given by*

$$(f^*S)_\Sigma \;=\; \{\langle \sigma, \phi \rangle \in D \times E \mid f_1(\sigma)\downarrow \vee f_2(\phi)\downarrow \implies \langle f_1(\sigma), f_2(\phi) \rangle \in S_\Sigma\}$$

for all morphisms $f = \langle f_1, f_2 \rangle$ with $f_1 : D \rightharpoonup D'$ and $f_2 : E \rightharpoonup E'$ and $S \in \mathcal{R}(D', E')$; and $\mathcal{R}$ possesses intersections given componentwise by set-theoretic intersection. Moreover, $\mathcal{R}_{adm} = \mathcal{R}$.

Next, we observe that Φ indeed maps admissible predicates over $\mathsf{St} \times \mathsf{RSF}$ to admissible predicates.

Lemma 5.1.10. *Let $X, Y \in \mathcal{R}(\mathsf{St}, \mathsf{RSF})$. Then*

$$\Phi(Y,X)_\Sigma \in \mathcal{A}dm(F_{Store}(\mathsf{St}, \mathsf{St}) \times F_{\mathsf{RSF}}(\mathsf{St}, \mathsf{RSF}, \mathsf{RSF}))$$

for all $\Sigma \in StSpec$.

Proof. To show closure under taking least upper bounds, suppose $\langle \sigma_0, \phi_0 \rangle \sqsubseteq \langle \sigma_1, \phi_1 \rangle \sqsubseteq \ldots$ is a chain in $\Phi(Y,X)_\Sigma$. Thus, $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \ldots$ is a chain in $F_{Store}(\mathsf{St}, \mathsf{St})$ and $\phi_0 \sqsubseteq \phi_1 \sqsubseteq \ldots$ is a chain in $F_{\mathsf{RSF}}(\mathsf{St}, \mathsf{RSF}, \mathsf{RSF})$. Let $\sigma \stackrel{def}{=} \bigsqcup_k \sigma_k$ and $\phi \stackrel{def}{=} \bigsqcup_k \phi_k$ (so $\langle \sigma, \phi \rangle = \bigsqcup_k \langle \sigma_k, \phi_k \rangle$), we show $\langle \sigma, \phi \rangle \in \Phi(Y,X)_\Sigma$ under the assumption that $X_{\Sigma'}$ is admissible for all $\Sigma' \in StSpec$.

Clearly the first condition (DOM) of the definition in Table 5.2, concerning the domains, is satisfied since $\mathsf{dom}(\phi) = \mathsf{dom}(\phi_k)$ for all $k \in \mathbb{N}$. As for the remaining conditions, suppose $l \in \mathsf{dom}(\Sigma)$ with $\Sigma.l \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$. Since, for all $i \in I$,

$$\sigma_0.l.\mathsf{f}_i \;=\; \sigma_1.l.\mathsf{f}_i \;=\; \ldots \;=\; \sigma.l.\mathsf{f}_i$$

we obtain $\sigma.l.\mathsf{f}_i \in \|A_i\|_\Sigma$ by assumption $\langle \sigma_0, \phi_0 \rangle \in \Phi(Y,X)_\Sigma$.

Next, suppose $\Sigma' \succcurlyeq \Sigma$, $\langle \sigma', \phi' \rangle \in Y_{\Sigma'}$, $l' \in \|\Sigma.l\|_{\Sigma'}$, $v \in \mathsf{Val}$ and $\sigma'' \in \mathsf{St}$ such that $\sigma.l.\mathsf{m}_j(l', \sigma') = \langle v, \sigma'' \rangle$. By definition of σ as $\bigsqcup_k \sigma_k$ and continuity, for all sufficiently

large k we must have $\sigma_k.l.\mathsf{m}_j(l',\sigma')\downarrow$. That is, there are $\sigma_k'' \in \mathsf{St}$ such that $\sigma_k.l.\mathsf{m}_j(l',\sigma') = \langle v, \sigma_k''\rangle$ for sufficiently large k, and

$$\langle v, \sigma''\rangle = \bigsqcup_k \sigma_k.l.\mathsf{m}_j(\sigma') = \bigsqcup_k \langle v, \sigma_k''\rangle$$

By assumption $\langle \sigma_0, \phi_0\rangle \sqsubseteq \langle \sigma_1, \phi_1\rangle \sqsubseteq \ldots$ in $\Phi(Y,X)_\Sigma$, for all sufficiently large k there are $\phi_k'' \in \mathsf{RSF}$ and $\Sigma_k'' \in \mathit{StSpec}$ such that

$$\phi_k.l.\mathsf{m}_j(l',\sigma',\phi',\Sigma') = \langle \Sigma_k'', \phi_k''\rangle$$

with $\Sigma_k'' \succcurlyeq \Sigma'$ and

(M1') $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma_k'')) \in [\![T_j[l'/y_j]]\!]$;
(M2') $\langle \sigma_k'', \phi_k''\rangle \in X_{\Sigma_k''}$; and
(M3') $v \in \left|\left|B_j[l'/y_j]\right|\right|_{\Sigma_k''}$

Since $\pi_{\mathrm{Val}}(\sigma_k'') = \pi_{\mathrm{Val}}(\sigma'')$ by (3.7), (M1) follows:

(M1) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l'/y_j]]\!]$

The discrete order on *Spec* entails $\Sigma_k'' \equiv \Sigma_{k+1}'' \equiv \ldots$, hence, $\phi(l',\sigma',\phi',\Sigma') = \bigsqcup_k \langle \Sigma_k'', \phi_k''\rangle = \langle \Sigma'', \bigsqcup_k \phi_k''\rangle$ with $\Sigma'' \equiv \Sigma_k'' \equiv \Sigma_{k+1}'' \equiv \ldots$. Thus (M3') entails (M3):

(M3) $v \in \left|\left|B_j[l'/y_j]\right|\right|_{\Sigma''}$

By assumption, $X_{\Sigma''}$ is an admissible subset of $\mathsf{St} \times \mathsf{RSF}$. Therefore also condition (M2) holds as required:

(M2) $\langle \sigma'', \phi''\rangle = \bigsqcup_k \langle \sigma_k'', \phi_k''\rangle \in X_{\Sigma''}$

since $(\langle \sigma'', \phi''\rangle)_{k\in\mathbb{N}}$ is a countable chain in $X_{\Sigma''}$. □

Next we consider the locally continuous functor

$$F_{\mathsf{St},\mathsf{RSF}}(R,S) : (\mathbf{pCpo} \times \mathbf{pCpo})^{op} \times (\mathbf{pCpo} \times \mathbf{pCpo}) \longrightarrow \mathbf{pCpo} \times \mathbf{pCpo}$$

defined as

$$F_{\mathsf{St},\mathsf{RSF}}(D,E) \stackrel{def}{=} \langle F_{\mathit{Store}}(\Pi_1(D),\Pi_1(E)), F_{\mathsf{RSF}}(\Pi_1(D),\Pi_2(D),\Pi_2(E))\rangle$$

where $\Pi_i : \mathbf{pCpo} \times \mathbf{pCpo} \longrightarrow \mathbf{pCpo}$ is the projection functor to the i-th component. Note that the pair $\langle \mathsf{St}, \mathsf{RSF}\rangle$ is the minimal invariant for this functor. In the following, we write F_i for the functor $\Pi_i \circ F_{\mathsf{St},\mathsf{RSF}}$.

We show that Φ determines an admissible action of this functor, in the following sense:

Lemma 5.1.11. *Let $X, X', Y, Y' \in \mathcal{R}(\mathsf{St},\mathsf{RSF})$ and suppose $e \sqsubseteq \mathsf{id}_{\langle \mathsf{St},\mathsf{RSF}\rangle}$. Then,*

$$e : X \subset X' \ \wedge\ e : Y' \subset Y \quad \Longrightarrow \quad F_{\mathsf{St},\mathsf{RSF}}(e,e) : \Phi(Y,X) \subset \Phi(Y',X') \qquad (\clubsuit)$$

Proof. This follows from a similar line of reasoning as in (Reus and Streicher 2004): Suppose $e = \langle e_1, e_2 \rangle \sqsubseteq \mathsf{id}_{\langle \mathsf{St},\mathsf{RSF} \rangle}$ such that

$$e : X \subset X' \quad \wedge \quad e : Y' \subset Y \tag{5.5}$$

for some $X, Y, X', Y' \in \mathcal{R}(\mathsf{St}, \mathsf{RSF})$. Assume $\langle \sigma, \phi \rangle \in \Phi(Y, X)_\Sigma$. To prove the lemma we have to show $F_{\mathsf{St},\mathsf{RSF}}(e,e)(\sigma,\phi) \in \Phi(Y', X')_\Sigma$. Recall that from the action of the functor on morphisms,

$$\begin{aligned} F_1(e,e)(\sigma).l.\mathsf{f} &= F_{\mathsf{St}}(e_1,e_1)(\sigma).l.\mathsf{f} \;=\; \sigma.l.\mathsf{f} \\ F_1(e,e)(\sigma).l.\mathsf{m}(l',\sigma') &= F_{\mathsf{St}}(e_1,e_1)(\sigma).l.\mathsf{m}(l',\sigma') \\ &= (\mathsf{id}_{\mathsf{Val}} \times e_1)(\sigma.l.\mathsf{m}(l', e_1(\sigma'))) \\ F_2(e,e)(\phi).l.\mathsf{m}(l',\sigma',\phi',\Sigma') &= F_{\mathsf{RSF}}(e_1,e_2,e_2)(\phi).l.\mathsf{m}(l',\sigma',\phi',\Sigma') \\ &= (\mathsf{id}_{StSpec} \times e_2)(\phi.l.\mathsf{m}(l', e_1(\sigma'), e_2(\phi'), \Sigma')) \end{aligned} \tag{5.6}$$

for all $\mathsf{f} \in \mathcal{F}$ and $\mathsf{m} \in \mathcal{M}$. In particular, condition (DOM) of Definition 5.1.7 is satisfied for $F_{\mathsf{St},\mathsf{RSF}}(e,e)(\sigma,\phi)$.

To show the remaining conditions let $l \in \mathsf{dom}(\Sigma)$, and $[\mathsf{f}_i\colon A_i,\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J} \equiv \Sigma.l$. From $\langle \sigma, \phi \rangle \in \Phi(Y,X)_\Sigma$ and (5.6) we immediately obtain

(F) $\quad F_{\mathsf{St}}(e_1,e_1)(\sigma).l.\mathsf{f}_i \in \|A_i\|_\Sigma \quad \forall i \in I$

Now we check conditions (M1)–(M3) of Definition 5.1.7. Let $j \in J$. Suppose $\Sigma' \succcurlyeq \Sigma$, $\phi' \in \mathsf{RSF}$, $l' \in \mathsf{Loc}$ and $\sigma' \in \mathsf{St}$ with $\langle \sigma', \phi' \rangle \in Y'_{\Sigma'}$ and such that $F_{\mathsf{St}}(e_1,e_1)(\sigma).l.\mathsf{m}_j(l',\sigma') \downarrow$. By (5.6) we thus know that

$$\begin{aligned} F_{\mathsf{St}}(e_1,e_1)(\sigma).l.\mathsf{m}_j(l',\sigma') \;&=\; \langle \nu, e_1(\sigma'') \rangle \\ \text{where } \langle \nu, \sigma'' \rangle \;&\stackrel{def}{=}\; \sigma.l.\mathsf{m}_j(l', e_1(\sigma')) \end{aligned}$$

for $\nu \in \mathsf{Val}$ and $\sigma'' \in \mathsf{St}$. By assumption (5.5), assumption $\langle \sigma', \phi' \rangle \in Y'_{\Sigma'}$ shows $e(\sigma',\phi') = \langle e_1(\sigma'), e_2(\phi') \rangle \in Y_{\Sigma'}$. Together with the assumption $\langle \sigma, \phi \rangle \in \Phi(Y,X)_\Sigma$ and (5.6) this entails

$$\begin{aligned} F_{\mathsf{RSF}}(e_1,e_2,e_2)(\phi).l.\mathsf{m}_j(l',\sigma',\phi',\Sigma') \;&=\; \langle \Sigma'', e_2(\phi'') \rangle \\ \text{where } \langle \Sigma'', \phi'' \rangle \;&\stackrel{def}{=}\; \phi.l.\mathsf{m}_j(l', e_1(\sigma'), e_2(\phi'), \Sigma') \end{aligned}$$

by definition of Φ; where $\phi'' \in \mathsf{RSF}$ and $\Sigma'' \succcurlyeq \Sigma'$ satisfy

(M1') $\quad (\pi_{\mathrm{Val}}(e_1(\sigma')), \nu, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T[l'/y_j]]\!]$
(M2') $\quad \langle \sigma'', \phi'' \rangle \in X_{\Sigma''}$
(M3) $\quad \nu \in \left\| B_j[l'/y_j] \right\|_{\Sigma''}$

Since $e \sqsubseteq \mathsf{id}_{\langle \mathsf{St},\mathsf{RSF} \rangle}$ we know $e_1(\sigma'') \sqsubseteq \sigma''$, and in particular $\pi_{\mathrm{Val}}(e_1(\sigma'')) = \pi_{\mathrm{Val}}(\sigma'')$. Similarly, since $e_1(\sigma') \sqsubseteq \sigma'$ we have $\pi_{\mathrm{Val}}(e_1(\sigma')) = \pi_{\mathrm{Val}}(\sigma')$. Hence, (M1') entails

(M1) $\quad (\pi_{\mathrm{Val}}(\sigma'), \nu, \pi_{\mathrm{Val}}(e_1(\sigma''))) \in [\![T[l/y_j]]\!]$

Finally, assumption (5.5) and (M2') give $\langle e_1(\sigma''), e_2(\phi'')\rangle \in X'_{\Sigma''}$ which shows (M2), and we have proved (♣).

Note that it is necessary that the predicates denoting transition specifications are 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 (unless one finds an appropriate way to restrict the reference to methods in transitions specifications; see Reus and Streicher 2004). □

Proof of Lemma 5.1.8. By Lemma 5.1.9, $\mathcal{R}$ is a relational structure on the bilimit-compact category **pCpo** × **pCpo** with inverse images and intersections. Lemmas 5.1.10 and 5.1.11 show that Theorem 2.5.3 is applicable which proves the existence of the invariant $\mathcal{R}$-relation X for $F_{\mathsf{St},\mathsf{RSF}}$ with respect to the action Φ. In particular, $X = \Phi(X, X)$ holds by definition of the invariant $\mathcal{R}$-relation □

Let $\delta_{\mathsf{St}}(e) = F_{Store}(e, e)$ and define maps $\delta^n_{\mathsf{RSF}} : \mathsf{RSF} \to \mathsf{RSF}$ for all $n \in \mathbb{N}$, by $\delta^0_{\mathsf{RSF}} = \bot$ and

$$\delta^n_{\mathsf{RSF}}(\phi).l.\mathsf{m}(l', \sigma', \phi', \Sigma') = (id \times \delta^{n-1}_{\mathsf{RSF}})(\phi.l.\mathsf{m}(l', \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma'), \delta^{n-1}_{\mathsf{RSF}}(\phi), \Sigma'))$$

for $n > 0$. In particular, $\delta^n_{\mathsf{RSF}}(\phi) \sqsubseteq \phi$, for all $n \in \mathbb{N}$ and all $\phi \in \mathsf{RSF}$. We note the following property of the invariant relation $fix(\Phi)$, which will be essential in the proof of Lemma 5.2.1 below.

Lemma 5.1.12 (Closure under projections)**.** *Let* $X \stackrel{def}{=} fix(\Phi) \in \mathcal{R}(\mathsf{St}, \mathsf{RSF})$ *and assume* $\langle \sigma, \phi\rangle \in X_\Sigma$. *Then*

$$\delta^n_{\mathsf{St}}(\bot)(\sigma)\downarrow \quad \Longrightarrow \quad \delta^n_{\mathsf{RSF}}(\phi)\downarrow \ \wedge \langle \delta^n_{\mathsf{St}}(\bot)(\sigma), \delta^n_{\mathsf{RSF}}(\phi)\rangle \in X_\Sigma$$

for all $n \in \mathbb{N}$. *In particular, if* $\sigma \in [\![\Sigma]\!]$ *then* $\delta^n_{\mathsf{St}}(\bot)(\sigma) \in [\![\Sigma]\!]$, *for all sufficiently large* n.

Proof. The proof is by induction on n. We observe that $\delta^n_{\mathsf{St}}(\bot)(\sigma)\uparrow$ if and only if $\delta^n_{\mathsf{RSF}}(\phi)\uparrow$ if and only if $n = 0$, hence there is nothing to show in the case $n = 0$. So assume $n > 0$ and note that by the above observation both $\delta^n_{\mathsf{St}}(\bot)(\sigma)\downarrow$ and $\delta^n_{\mathsf{RSF}}(\phi)\downarrow$ in this case. We show that $\langle \delta^n_{\mathsf{St}}(\bot)(\sigma), \delta^n_{\mathsf{RSF}}(\phi)\rangle \in X_\Sigma$.

The conditions (DOM) obviously hold, since by assumption $\langle \sigma, \phi\rangle \in X_\Sigma$ and $\delta^n_{\mathsf{RSF}}(\phi) \sqsubseteq \phi$ entails $\mathsf{dom}(\delta^n_{\mathsf{RSF}}(\phi)) = \mathsf{dom}(\phi)$ and $\mathsf{dom}(\delta^n_{\mathsf{RSF}}(\phi).l) = \mathsf{dom}(\phi.l)$ for all $l \in \mathsf{dom}(\Sigma)$.

Let $l \in \mathsf{dom}(\Sigma)$ and suppose $\Sigma.l$ is $[\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}$. Since $\delta^n_{\mathsf{St}}(\bot)(\sigma) \sqsubseteq \sigma$ implies $\delta^n_{\mathsf{St}}(\bot)(\sigma).l.\mathsf{f}_i = \sigma.l.\mathsf{f}_i$ for all $i \in I$, condition (F) is satisfied. So let $j \in J$, $\Sigma' \succcurlyeq \Sigma$, $l' \in \|\Sigma.l\|_{\Sigma'}$ and $\langle \sigma', \phi'\rangle \in X_{\Sigma'}$, and suppose $\delta^n_{\mathsf{St}}(\bot)(\sigma).l.\mathsf{m}_j(l', \sigma')\downarrow$, i.e., $\delta^n_{\mathsf{St}}(\bot)(\sigma).l.\mathsf{m}_j(l', \sigma') = \langle \nu, \sigma''\rangle$ for some ν and σ''. By definition of δ_{St} there exists $\hat\sigma$ with $\sigma.l.\mathsf{m}_j(l', \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma')) = \langle \nu, \hat\sigma\rangle$ and $\sigma'' = \delta^{n-1}_{\mathsf{St}}(\bot)(\hat\sigma)$. In particular, $\delta^{n-1}_{\mathsf{St}}(\bot)(\sigma')\downarrow$ so that the induction hypothesis yields $\delta^{n-1}_{\mathsf{RSF}}(\phi')\downarrow$ and $\langle \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma'), \delta^{n-1}_{\mathsf{RSF}}(\phi')\rangle \in X_{\Sigma'}$. By $\langle \sigma, \phi\rangle \in X_\Sigma$ there exist $\Sigma'' \succcurlyeq \Sigma'$ and $\hat\phi$ such that $\phi.l.\mathsf{m}_j(l', \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma'), \delta^{n-1}_{\mathsf{RSF}}(\phi'), \Sigma') = \langle \Sigma'', \hat\phi\rangle$ and (M1)–(M3) hold, i.e., $(\pi_{\mathrm{Val}}(\delta^{n-1}_{\mathsf{RSF}}(\sigma')), \nu, \pi_{\mathrm{Val}}(\hat\sigma)) \in [\![T_j[l'/y_j]]\!]$ and $\langle \hat\sigma, \hat\phi\rangle \in X_{\Sigma''}$ and $\nu \in \left|\left|B_j[l'/y_j]\right|\right|_{\Sigma''}$.

Since $\delta_{\mathsf{St}}^{n-1}(\bot)(\hat{\sigma}) = \sigma''$ is defined, the induction hypothesis applies and we obtain $\langle \sigma'', \phi'' \rangle = \langle \delta_{\mathsf{St}}^{n-1}(\bot)(\hat{\sigma}), \delta_{\mathsf{RSF}}^{n-1}(\hat{\phi}) \rangle \in X_{\Sigma''}$, for $\phi'' = \delta_{\mathsf{RSF}}^{n-1}(\hat{\phi})$. Thus, by definition of $\delta_{\mathsf{RSF}}^{n}$,

$$\delta_{\mathsf{RSF}}^{n}(\phi).l.\mathsf{m}(l', \sigma', \phi', \Sigma') = (id \times \delta_{\mathsf{RSF}}^{n-1})(\phi.l.\mathsf{m}(l', \delta_{\mathsf{St}}^{n-1}(\bot)(\sigma'), \delta_{\mathsf{RSF}}^{n-1}(\phi), \Sigma')) = \langle \Sigma'', \phi'' \rangle$$

so that (M1) holds also for $\langle \delta_{\mathsf{St}}^{n}(\bot)(\sigma), \delta_{\mathsf{RSF}}^{n}(\phi) \rangle$. Further, note that $\delta_{\mathsf{St}}^{n-1}(\bot)(\sigma') \sqsubseteq \sigma'$ and $\sigma'' = \delta_{\mathsf{St}}^{n-1}(\bot)(\hat{\sigma}) \sqsubseteq \hat{\sigma}$ yields $(\pi_{\mathrm{Val}}(\sigma'), \nu, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l'/y_j]]\!]$, i.e., (M2) holds. Finally, (M3) holds since $\langle \sigma'', \phi'' \rangle \in X_{\Sigma''}$.

For the final assertion of the lemma observe that $id_{\mathsf{St}} = lfp(\delta_{\mathsf{St}})$ by the minimal invariant property. By continuity, $\delta_{\mathsf{St}}^{n}(\bot)(\sigma)\downarrow$ for all $\sigma \in \mathsf{St}$ and all sufficiently large $n \in \mathbb{N}$. □

5.2 Soundness

5.2.1 Preliminaries

Recall from the previous section that the semantics of store specifications is defined in terms of the semantics $\|A\|_\Sigma$ for result specifications A. This semantics does not mention St at all. The following key lemma establishes the relation between the object and store specifications of Section 5.1, and object specifications $[\![A]\!]$ as defined in Section 4.3.4:

Lemma 5.2.1. *For all object specifications A, store specifications $\Sigma \in StSpec$, stores $\sigma \in \mathsf{St}$, and locations $l \in \mathsf{Loc}$,*

$$\sigma \in [\![\Sigma]\!] \;\wedge\; l \in \|A\|_\Sigma \quad \Longrightarrow \quad \langle l, \sigma \rangle \in [\![A]\!]$$

Proof. We prove the following property.

Claim. For all $n \in \mathbb{N}$, for all object specifications $A \in Spec$, for all $\Sigma \in StSpec$, for all $l \in \mathsf{dom}(\Sigma)$ and all $\sigma \in [\![\Sigma]\!]$,

$$l \in \|A\|_\Sigma \;\wedge\; \delta_{\mathsf{St}}^{n}(\bot)(\sigma)\downarrow \quad \Longrightarrow \quad \langle l, \delta_{\mathsf{St}}^{n}(\bot)(\sigma) \rangle \in [\![A]\!] \tag{5.7}$$

From this claim the lemma follows, since for all $\sigma \in \mathsf{St}$, $\delta_{\mathsf{St}}^{n}(\bot)(\sigma)\downarrow$ for almost all $n \in \mathbb{N}$, $\sigma = \bigsqcup_n \delta_{\mathsf{St}}^{n}(\bot)(\sigma)$, and $[\![A]\!]$ is closed under taking least upper bounds by Lemma 4.3.5.

The proof of the claim proceeds by induction on $n \in \mathbb{N}$ and $A \in Spec$, using the well-founded lexicographic order on $\mathbb{N} \times Spec$ where specifications are ordered according to size. Since $\delta_{\mathsf{St}}^{0}(\bot)(\sigma)\uparrow$ there is nothing to show in the case $n = 0$. So assume $n > 0$, $\delta_{\mathsf{St}}^{n}(\bot)(\sigma)\downarrow$ and write $\sigma_n \stackrel{def}{=} \delta_{\mathsf{St}}^{n}(\bot)(\sigma)$.

Note that because A is an object specification it is necessarily of the form

$$A \quad \equiv \quad [\mathsf{f}_i\colon A_i,\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: T_j]_{i \in I, j \in J}$$

We prove $\langle l, \sigma_n \rangle \in [\![A]\!]$, i.e., that

(i) for all $i \in I$, $\langle \sigma_n.l.\mathsf{f}_i, \sigma_n \rangle \in [\![A_i]\!]$; and

(ii) for all $j \in J$, if there are $v \in \mathsf{Val}$ and $\sigma' \in \mathsf{St}$ such that $\sigma_n.l.\mathsf{m}_j(l, \sigma_n) = \langle v, \sigma' \rangle$, then

- $\langle v, \sigma' \rangle \in [\![y_j \vdash B_j]\!]\,(y_j \mapsto l)$; and
- $(\pi_{\mathrm{Val}}(\sigma_n), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![y_j \vdash T_j]\!]\,(y_j \mapsto l)$.

Recall that by definition, $l \in \|A\|_\Sigma$ means $\Sigma.l \prec: A$. From the subtyping relation we therefore find

$$\Sigma.l \quad \equiv \quad [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B'_j{::}T'_j]_{i\in I', j\in J'}$$

where $I \subseteq I'$, $J \subseteq J'$, and $y_j \vdash B'_j \prec: B_j$ and $y_j \vdash_{\mathsf{fo}} T'_j \to T_j$ for all $j \in J$.

For the first part, note that Lemma 5.1.12 and $\sigma \in [\![\Sigma]\!]$ entails $\sigma_n = \delta^n_{\mathsf{St}}(\bot)(\sigma) \in [\![\Sigma]\!]$. By Definition 5.1.7 part (F) and $\sigma_n \in [\![\Sigma]\!]$ we have $\sigma_n.l.\mathsf{f}_i \in \|A_i\|_\Sigma$ for all $i \in I'$. If A_i is bool then $\|\mathsf{bool}\|_\Sigma = \mathsf{BVal}$, hence, $\langle \sigma_n.l.\mathsf{f}_i, \sigma_n\rangle \in \mathsf{BVal} \times \mathsf{St} = [\![\mathsf{bool}]\!]$. Otherwise A_i is an object specification strictly smaller than A and we may apply the induction hypothesis (on n, A_i, Σ, $\sigma_n.l.\mathsf{f}_i$ and $\sigma \in [\![\Sigma]\!]$) to conclude $\langle \sigma_n.l.\mathsf{f}_i, \sigma_n\rangle \in [\![A_i]\!]$.

For the second part, let $j \in J$ and suppose that $\sigma_n.l.\mathsf{m}_j(l, \sigma_n) = \langle \nu, \sigma''\rangle$ for some ν and σ''. By definition of the projections δ^n_{St}, this means that $\sigma.l.\mathsf{m}_j(l, \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma_n)) = \langle \nu, \hat\sigma\rangle$ for some $\hat\sigma$ such that $\delta^{n-1}_{\mathsf{St}}(\bot)(\hat\sigma) = \sigma''$. In particular, $\delta^{n-1}_{\mathsf{St}}(\bot)(\sigma_n)\downarrow$ and $\delta^{n-1}_{\mathsf{St}}(\bot)(\hat\sigma)\downarrow$. We observe that $\delta^{n-1}_{\mathsf{St}}(\bot)(\sigma_n) = \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma)$, since by Lemma 2.3.3 one has $\delta^m_{\mathsf{St}}(\bot) \circ \delta^k_{\mathsf{St}}(\bot) = \delta^m_{\mathsf{St}}(\bot)$ whenever $m \le k$. Thus, $\delta^{n-1}_{\mathsf{St}}(\bot)(\sigma)\downarrow$ and Lemma 5.1.12 yields $\sigma_{n-1} \in [\![\Sigma]\!]$, where we write $\sigma_{n-1} \stackrel{def}{=} \delta^{n-1}_{\mathsf{St}}(\bot)(\sigma)$. Hence, from Definition 5.1.7 parts (M2) and (M3), the fact that $l \in \|\Sigma.l\|_\Sigma$ and that $\sigma \in [\![\Sigma]\!]$ we find $\nu \in \left|\left|B'_j[l/y_j]\right|\right|_{\Sigma''}$ and $\hat\sigma \in [\![\Sigma'']\!]$ for some $\Sigma'' \succcurlyeq \Sigma$.

In the case where B_j (and thus also B'_j) is bool we therefore have $\nu \in \mathsf{BVal}$ and immediately obtain

$$\langle \nu, \sigma''\rangle \in \mathsf{BVal} \times \mathsf{St} = [\![\mathsf{bool}]\!] = [\![y_j \vdash \mathsf{bool}]\!]\,(y_j \mapsto l)$$

In the remaining case where B_j (and thus also B'_j) is an object specification then by induction hypothesis (applied to $n-1$, $B'_j[l/y_j]$, Σ'', $\hat\sigma \in [\![\Sigma'']\!]$ and ν) this yields $\langle \nu, \sigma''\rangle \in [\![B'_j[l/y_j]]\!]$, observing that $\sigma'' = \delta^{n-1}_{\mathsf{St}}(\bot)(\hat\sigma)$. Thus,

$$\begin{aligned} \langle \nu, \sigma''\rangle \in [\![B'_j[l/y_j]]\!] &= [\![y_j \vdash B'_j]\!]\,(y_j \mapsto l) && \text{by Lemma 5.1.2} \\ &\subseteq [\![y_j \vdash B_j]\!]\,(y_j \mapsto l) && \text{by Lemma 4.3.6} \end{aligned}$$

as required.

Similarly, by Definition 5.1.7 (M1) we obtain

$$\begin{aligned} (\pi_{\mathrm{Val}}(\sigma_{n-1}), \nu, \pi_{\mathrm{Val}}(\hat\sigma)) \in [\![T'_j[l/y_j]]\!] &= [\![y_j \vdash T'_j]\!]\,(y_j \mapsto l) && \text{by Lemma 5.1.2} \\ &\subseteq [\![y_j \vdash T_j]\!]\,(y_j \mapsto l) && \text{soundness of } \vdash_{\mathsf{fo}} \end{aligned}$$

Since $\sigma_{n-1} \sqsubseteq \sigma_n$ and $\sigma'' = \delta^{n-1}_{\mathsf{St}}(\bot)(\hat\sigma) \sqsubseteq \hat\sigma$ implies $\pi_{\mathrm{Val}}(\sigma_n) = \pi_{\mathrm{Val}}(\sigma_{n-1})$ and $\pi_{\mathrm{Val}}(\sigma'') = \pi_{\mathrm{Val}}(\hat\sigma)$, we also have

$$(\pi_{\mathrm{Val}}(\sigma_n), \nu, \pi_{\mathrm{Val}}(\sigma'')) \in [\![y_j \vdash T_j]\!]\,(y_j \mapsto l)$$

This concludes the proof. □

Remark 5.2.2. *The use of a lexicographic order in the proof of Lemma 5.2.1 highlights the fact that one has to deal with* two *problems here: possibly cyclic pointer structures in the*

heap (realised by the field values), and recursive methods. Due to the invariance in field specifications, induction on A can be used to deal with the former while the projections $\delta^n_{\mathsf{St}}(\bot)$ resolve the latter problem. Note that a simple induction on the size of A alone is not sufficient, because by subsumption in the case of method specifications the induction hypothesis is, in general, not applicable.[1]

We can now define the semantics of judgements of Abadi-Leino logic and prove the key lemma. The semantics of a judgement is written $\Gamma \vDash a : A :: T$ and states the following: Suppose (the denotation of) an object term $[\![a]\!]$ returns (v, σ') when run in a store σ that satisfies some store specification Σ (therefore not containing any code that violates the restrictions needed for the logic, see also the discussion in Section 5.1) and in an environment ρ that satisfies Γ with respect to Σ. Then (v, σ') satisfies specification $[\![[\Gamma] \vdash A]\!]\,\rho$ and the state transformation provoked satisfies $[\![[\Gamma] \vdash T]\!]\,\rho$. This is spelt out formally below:

Definition 5.2.3 (Validity)**.** *$\Gamma \vDash a : A :: T$ if and only if for all $\Sigma \in$ StSpec, for all $\rho \in \|\Gamma\|_\Sigma$, for all $\sigma, \sigma' \in \mathsf{St}$ and for all $v \in \mathsf{Val}$,*

$$\sigma \in [\![\Sigma]\!] \;\wedge\; [\![a]\!]\,\rho\sigma = \langle v, \sigma'\rangle \quad\Longrightarrow\quad \langle v, \sigma'\rangle \in [\![[\Gamma] \vdash A]\!]\,\rho \;\wedge\; (\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T]\!]\,\rho$$

Unfortunately, this definition relies on the (syntactic) notion of store specifications to keep track of the specifications for heap objects that may be referenced from the environment ρ. We found the use of store specifications necessary because of the dynamic aspect of a store, for two reasons: first, locations may be updated, and some information is needed in order to *preserve* the specification of locations. Second, the store may *grow* during computation, which cannot be reflected by a fixed context Γ. Note that store specification extension is defined precisely to cover these two cases.

We believe this approach is justified, however, since for the case of closed programs $\Gamma \vdash a : A :: T$, i.e., where $\Gamma = \emptyset$, the condition $\rho \in \|\Gamma\|_\Sigma$ is vacuous and one obtains a "proper" semantic notion of validity.

Before proving the main technical result in Lemma 5.2.5 we state the following fact about the transition relation that appears in the let rule:

Lemma 5.2.4. *Suppose that for $\sigma, \sigma', \sigma'' \in \mathsf{St}$ and $v, v' \in \mathsf{Val}$, $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x} \vdash T']\!]\,\rho$ and $(\pi_{\mathrm{Val}}(\sigma'), v', \pi_{\mathrm{Val}}(\sigma'')) \in [\![\overline{x}, x \vdash T'']\!]\,\rho[x := v]$. Then, if $\overline{x} \vdash T$ and*

$$\begin{aligned} &\vdash_{\mathsf{fo}} T'[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{post}(\cdot), x/\mathsf{result}] \\ &\quad \wedge\; T''[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{pre}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathit{alloc}_{pre}(\cdot)] \to T \end{aligned} \tag{5.8}$$

then $(\pi_{\mathrm{Val}}(\sigma), v', \pi_{\mathrm{Val}}(\sigma'')) \in [\![\overline{x} \vdash T]\!]\,\rho$.

Proof. Consider an extended signature of transition relations, with additional function and predicate symbols $\mathsf{sel}_{int}(\cdot,\cdot)$ and $\mathsf{alloc}_{int}(\cdot)$, resp. We extend the interpretation of transition relations in the natural way,

$$[\![x_1, \ldots, x_k \vdash T]\!]\,\rho : \mathcal{P}(\mathsf{St}_{\mathsf{Val}} \times \mathsf{Val} \times \mathsf{St}_{\mathsf{Val}} \times \mathsf{St}_{\mathsf{Val}})$$

[1] A faulty proof attempt along these lines was given in a previous version of this work. Thanks to Guy McCusker for pointing out the problem.

where the second store argument is used to interpret $\mathsf{sel}_{int}(\cdot,\cdot)$ and $\mathsf{alloc}_{int}(\cdot)$:

$$(\sigma, v, \sigma', \sigma'') \in [\![\overline{x} \vdash \mathsf{alloc}_{int}(e)]\!]\,\rho \quad \overset{def}{\iff} \quad [\![\overline{x} \vdash e]\!]\,\rho\sigma v\hat{\sigma}\sigma' \downarrow \ \wedge\ [\![\overline{x} \vdash e]\!]\,\rho\sigma v\hat{\sigma}\sigma' \in \mathsf{dom}(\hat{\sigma})$$

and

$$[\![\overline{x} \vdash \mathsf{sel}_{int}(e_0, e_1)]\!]\,\rho\sigma v\hat{\sigma}\sigma' \;=\; \hat{\sigma}.l.\mathsf{f}$$

if $[\![\overline{x} \vdash e_0]\!]\,\rho\sigma v\hat{\sigma}\sigma' = l \in \mathsf{Loc}$ and $[\![\overline{x} \vdash e_1]\!]\,\rho\sigma v\hat{\sigma}\sigma' = \mathsf{f} \in \mathcal{F}$; undefined otherwise.

By assumption and using the fact that neither T' nor T'' contains the new predicates, we also have

$$(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma'), \pi_{\mathrm{Val}}(\sigma')) \in [\![\overline{x}, x \vdash T']\!]\,\rho[x := v]$$

and

$$(\pi_{\mathrm{Val}}(\sigma'), v', \pi_{\mathrm{Val}}(\sigma'), \pi_{\mathrm{Val}}(\sigma'')) \in [\![\overline{x}, x \vdash T'']\!]\,\rho[x := v]$$

Thus,

$$\begin{aligned}&(\pi_{\mathrm{Val}}(\sigma), v', \pi_{\mathrm{Val}}(\sigma'), \pi_{\mathrm{Val}}(\sigma'')) \in \\ &\quad [\![\overline{x}, x \vdash T'[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{post}(\cdot), x/\mathsf{result}]]\!]\,\rho[x := v]\end{aligned}$$

since there are no occurrences of $\mathsf{sel}_{post}(\cdot,\cdot)$, $\mathsf{alloc}_{post}(\cdot)$ and result, and

$$\begin{aligned}&(\pi_{\mathrm{Val}}(\sigma), v', \pi_{\mathrm{Val}}(\sigma'), \pi_{\mathrm{Val}}(\sigma'')) \in \\ &\quad [\![\overline{x}, x \vdash T''[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{pre}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{pre}(\cdot)]]\!]\,\rho[x := v]\end{aligned}$$

since there are no occurrences of $\mathsf{sel}_{pre}(\cdot,\cdot)$ and $\mathsf{alloc}_{pre}(\cdot)$. From validity of first-order provability and assumption (5.8) we obtain

$$(\pi_{\mathrm{Val}}(\sigma), v', \pi_{\mathrm{Val}}(\sigma'), \pi_{\mathrm{Val}}(\sigma'')) \in [\![\overline{x}, x \vdash T]\!]\,\rho[x := v]$$

and the result follows since T does not depend on x and the new predicates, by $\overline{x} \vdash T$. □

5.2.2 The Invariance Lemma

In this subsection we state and prove the main lemma of the soundness proof. Intuitively, it shows that store specifications Σ are "invariant" under proved programs,

$$\sigma \in [\![\Sigma]\!] \ \wedge\ [\![a]\!]\,\rho\sigma = \langle v, \sigma'\rangle \quad \Longrightarrow \quad \exists \Sigma' \in \mathit{StSpec}.\ \Sigma' \succcurlyeq \Sigma \ \wedge\ \sigma' \in [\![\Sigma']\!] \tag{5.9}$$

Note that the program a will in general allocate further objects, so the resulting store only satisfies an extension of the original store specification, not necessarily the orignial one. The precise conditions of when (5.9) holds are given in the statement of the following lemma, and take the choice functions $\phi \in \mathsf{RSF}$ introduced in Section 5.1 into account. We write SF for the domain of "individual" choice functions,

$$\mathsf{SF} \quad \overset{def}{=} \quad [\mathsf{Loc} \times \mathsf{St} \times \mathsf{RSF} \times \mathit{StSpec} \rightharpoonup \mathit{StSpec} \times \mathsf{RSF}]$$

for which $\mathsf{RSF} = \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{SF}))$.

Lemma 5.2.5. *Suppose*

(H1) $\Gamma \vdash a : A :: T$ *is derivable;*

(H2) $\Sigma \in \mathit{StSpec}$ *is a store specification; and*

(H3) $\rho \in \|\Gamma\|_\Sigma$.

Then there exists $\phi \in \mathsf{SF} = [\mathsf{Val} \times \mathsf{St} \times \mathsf{RSF} \times \mathit{StSpec} \rightharpoonup \mathit{StSpec} \times \mathsf{RSF}]$ *such that for all* $\Sigma' \succcurlyeq \Sigma$, *for all* $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, *for all* $l \in \mathsf{Loc}$ *and all* $\sigma'' \in \mathsf{St}$, *if*

$$[\![a]\!]\, \rho \sigma' \;=\; \langle v, \sigma'' \rangle$$

then the following holds:

(S1) *there exists* $\Sigma'' \succcurlyeq \Sigma'$ *and* $\phi'' \in \mathsf{RSF}$ *such that* $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$

(S2) $\langle \sigma'', \phi'' \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$

(S3) $v \in \|A[\rho/\Gamma]\|_{\Sigma''}$

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T]\!]\, \rho$

Proof. The proof is by induction on the derivation of $\Gamma \vdash a : A :: T$. In the proof,

- Lemma 5.1.6 is applied in the cases (AL LET) and (AL OBJ), where an extended specification context is used in the induction hypothesis;
- invariance of subspecifications in field specifications is needed in the case for rule (AL UPD);
- in the cases where the store changes, i.e., (AL OBJ), (AL UPD) and (AL COPY), we must show explicitly that the resulting store satisfies the store specification, according to Definition 5.1.7.

We consider cases, depending on the last rule applied in the derivation of the judgement $\Gamma \vdash a : A :: T$.

- Case (AL SUB)
  Suppose that $\Gamma \vdash a : A :: T$ has been obtained by an application of the subsumption rule, and that

  (H2) Σ is a store specification

  (H3) $\rho \in \|\Gamma\|_\Sigma$

  We have to show that there is $\phi \in \mathsf{SF}$ such that, whenever $\Sigma' \succcurlyeq \Sigma$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, $v' \in \|A\|_{\Sigma'}$ and $[\![a]\!]\, \rho \sigma' = \langle v, \sigma' \rangle$, then *(S1)–(S4)* hold.

  Recall the subsumption rule,

  $$\text{(AL SUB)} \qquad \frac{[\Gamma] \vdash A' \prec: A \quad \Gamma \vdash a{:}A' :: T' \quad [\Gamma] \vdash A \quad [\Gamma] \vdash T \quad \vdash_{\mathsf{fo}} T' \to T}{\Gamma \vdash a{:}A :: T}$$

  so we must have $\Gamma \vdash a : A' :: T'$ for some specification A' and transition relation T' with $\vdash_{\mathsf{fo}} T' \to T$ and $[\Gamma] \vdash A' \prec: A$.

  By (IH) there exists $\phi \in \mathsf{SF}$ such that for all $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ with $[\![a]\!]\, \rho \sigma' = \langle v, \sigma' \rangle$,

(S1) there exists $\Sigma'' \succcurlyeq \Sigma'$, $\phi'' \in \mathsf{RSF}$ s.t. $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$

(S2) $\langle \sigma'', \phi'' \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$

(S3') $v \in \|A'[\rho/\Gamma]\|_{\Sigma'}$

(S4') $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T']\!]\,\rho$

Because $\vdash_{\mathsf{fo}} T' \to T$ we know $[\![\Gamma \vdash T']\!]\,\rho \subseteq [\![\Gamma \vdash T]\!]\,\rho$, and therefore *(S4')* implies

$$(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![\Gamma \vdash T]\!]\,\rho \qquad (S4)$$

It remains to show

$$v \in \|A[\rho/\Gamma]\|_{\Sigma'} \qquad (S3)$$

Note that by the subtyping rules, $A \equiv \mathsf{bool}$ if and only if $A' \equiv \mathsf{bool}$. In this case *(S3)* follows directly from *(S3')*. In the case where A' is an object specification, assumption $[\Gamma] \vdash A' \prec: A$ and Lemma 5.1.2 entail $\vdash A'[\rho/\Gamma] \prec: A[\rho/\Gamma]$. Transitivity of $\prec:$ and *(S3')* then prove *(S3)*, by the definition of $\|A'[\rho/\Gamma]\|_{\Sigma'}$.

· Case (AL VAR)
Suppose $\Gamma \vdash a : A :: T$ has been derived by an application of the (AL VAR) rule. Further, assume

(H2) Σ is a store specification

(H3) $\rho \in \|\Gamma\|_{\Sigma}$

Define the (partial continuous) map $\phi \in \mathsf{SF}$ by

$$\phi(l, \sigma', \phi', \Sigma') \quad \stackrel{def}{=} \quad \langle \Sigma', \phi' \rangle$$

Now suppose $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and $[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle$ Then, by the variable rule, we find that a is necessarily a variable x. Further we obtain $x{:}A$ in Γ, $T \equiv T_{\mathsf{res}}(x)$, and the semantics of Table 3.7 gives $\langle v, \sigma'' \rangle = [\![a]\!]\,\rho\sigma' = \langle \rho(x), \sigma' \rangle$, i.e.,

$$v = \rho(x) \quad \text{and} \quad \sigma'' = \sigma'$$

By definition of ϕ above,

(S1) $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma', \phi' \rangle$

(S2) $\langle \sigma'', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, by $\sigma'' = \sigma'$ and assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$

(S3) $v \in \|A[\rho/\Gamma]\|_{\Sigma'}$, by $v = \rho(x)$ and *(H3)*

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T_{\mathsf{res}}(x)]\!]\,\rho$, by the definition of $[\![[\Gamma] \vdash T]\!]$ in Table 4.6, and $\sigma'' = \sigma'$ and $v = \rho(x)$.

as required.

· Case (AL CONST)
Similar to the previous case.

· Case (AL COND)
By a case distinction, depending on whether the denotation of the guard x is *true* or *false*. We omit this case.

· Case (AL LET)
Suppose *(H1)* $\Gamma \vdash a : A :: T$ has been derived by an application of the (AL LET) rule. Hence, a is of the form $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$. Assume that

(H2) Σ is a store specification, and

(H3) $\rho \in \|\Gamma\|_\Sigma$

Now recall the rule for this case,

$$\text{(AL LET)}\quad \frac{\begin{array}{c}\Gamma \vdash a_1{:}A_1{::}T_1 \quad \Gamma, x{:}A_1 \vdash a_2{:}A{::}T_2 \quad [\Gamma] \vdash A \quad [\Gamma] \vdash T \\ \vdash_{\mathsf{fo}} T_1[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{post}(\cdot), x/\mathsf{result}] \\ \wedge\, T_2[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{pre}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{pre}(\cdot)] \to T\end{array}}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2{:}A{::}T}$$

By the premiss of this rule we must have

(H1') $\Gamma \vdash a_1 : A_1 :: T_1$

(H1") $\Gamma, x{:}A_1 \vdash a_2 : A :: T_2$

By induction hypothesis applied to *(H1')* there is $\phi_1 \in \mathsf{SF}$ such that for all $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ with $[\![a_1]\!]\,\rho\sigma' = \langle \hat{v}, \hat{\sigma} \rangle$, the conclusions of the lemma hold:

(S1') there exists $\hat{\Sigma} \succcurlyeq \Sigma'$ and $\hat{\phi} \in \mathsf{RSF}$ s.t. $\phi_1(l, \sigma', \phi', \Sigma') = \langle \hat{\Sigma}, \hat{\phi} \rangle$

(S2') $\langle \hat{\sigma}, \hat{\phi} \rangle \in \mathit{fix}(\Phi)_{\hat{\Sigma}}$

(S3') $\hat{v} \in \|A_1[\rho/\Gamma]\|_{\hat{\Sigma}}$

(S4') $(\pi_{\mathrm{Val}}(\sigma'), \hat{v}, \pi_{\mathrm{Val}}(\hat{\sigma})) \in [\![[\Gamma] \vdash T_1]\!]\,\rho$

In particular, by *(S3')* and Lemma 5.1.6,

$$\rho[x := \hat{v}] \in \|\Gamma, x{:}A_1\|_{\hat{\Sigma}}$$

Therefore, by induction hypothesis applied to *(H1")* there is $\phi_{\hat{\Sigma}\hat{v}} \in \mathsf{SF}$ s.t. for all $\Sigma' \succcurlyeq \hat{\Sigma}$, all $l' \in \mathsf{Loc}$ and all $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ with $[\![a_2]\!]\,\rho[x := \hat{v}]\sigma' = \langle v, \sigma'' \rangle$, the following holds:

(S1") there exists $\Sigma'' \succcurlyeq \Sigma'$ and $\phi'' \in \mathsf{RSF}$ s.t. $\phi_{\hat{\Sigma}\hat{v}}(l', \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$

(S2") $\langle \sigma'', \phi'' \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$

(S3") $v \in \|A[\rho[x := v]/\Gamma, x{:}A_1]\|_{\Sigma''}$

(S4") $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma, x{:}A_1] \vdash T_2]\!]\,\rho[x := \hat{v}]$

Now define $\phi \in \mathsf{SF}$ for all l, σ', ϕ' and Σ' by

$$\phi(l, \sigma', \phi', \Sigma') \quad \stackrel{def}{=} \quad \begin{cases} \phi_{\hat{\Sigma}\hat{v}}(l, \hat{\sigma}, \hat{\phi}, \hat{\Sigma}) & \text{if } [\![a_1]\!]\,\rho\sigma' = \langle \hat{v}, \hat{\sigma} \rangle \text{ and} \\ & \phi_1(l, \sigma', \phi', \Sigma') = \langle \hat{\Sigma}, \hat{\phi} \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

which is continuous due to continuity of $[\![a]\!]\,\rho$ and ϕ_1, and since Val and *StSpec* are discrete cpos.

We show that the conclusion of the lemma holds: Let $\Sigma' \succcurlyeq \Sigma$, let $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and $l \in \mathsf{Loc}$. Suppose $[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle$. From the definition of the semantics,

$$\langle v, \sigma'' \rangle \quad = \quad \textbf{let}\ \langle \hat{v}, \hat{\sigma} \rangle = [\![a_1]\!]\,\rho\sigma'\ \textbf{in}\ [\![a_2]\!]\,\rho[x := \hat{v}]\hat{\sigma}$$

which shows

- $[\![a_1]\!]\,\rho\sigma' = \langle \hat{v}, \hat{\sigma}\rangle$
- $[\![a_2]\!]\,\rho[x := \hat{v}]\hat{\sigma} = \langle v, \sigma''\rangle$

From the definition of ϕ, and the considerations above, it follows that

(S1) there is $\Sigma'' \in \mathit{StSpec}$ with $\Sigma'' \succcurlyeq \hat{\Sigma} \succcurlyeq \Sigma'$ and $\phi(l, \sigma', \phi', \Sigma') = \phi_{\hat{\Sigma}\hat{v}}(l, \hat{\sigma}, \hat{\phi}, \hat{\Sigma}) = \langle \Sigma'', \phi''\rangle$, where $\phi_1(l, \sigma', \phi', \Sigma') = \langle \hat{\Sigma}, \hat{\phi}\rangle$, by *(S1')* and *(S1")*

(S2) $\langle \sigma'', \phi''\rangle \in \mathit{fix}(\Phi)_{\Sigma''}$, by *(S2')* and *(S2")*

(C3) $v \in \|A[\rho[x := v]/\Gamma, x{:}A_1]\|_{\Sigma''}$, by *(S3')* and *(S3")*

(C4') $(\pi_{\mathrm{Val}}(\sigma'), \hat{v}, \pi_{\mathrm{Val}}(\hat{\sigma})) \in [\![[\Gamma] \vdash T_1]\!]\,\rho$, by *(S4')*

(C4") $(\pi_{\mathrm{Val}}(\hat{\sigma}), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma, x{:}A_1] \vdash T_2]\!]\,\rho[x := \hat{v}]$, by *(S4")*

Since $[\Gamma] \vdash A$, i.e., x is not free in A, we have

$$A[\rho[x := v]/(\Gamma, x{:}A_1)] \quad \equiv \quad A[\rho/\Gamma] \tag{5.10}$$

Moreover, *(C4')*, *(C4")*, Lemma 5.2.4 and

$$\begin{array}{l} \vdash_{\mathsf{fo}} T_1[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{post}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{post}(\cdot), x/\mathsf{result}] \\ \qquad \wedge\ T_2[\mathsf{sel}_{int}(\cdot,\cdot)/\mathsf{sel}_{pre}(\cdot,\cdot), \mathsf{alloc}_{int}(\cdot)/\mathsf{alloc}_{pre}(\cdot)] \ \rightarrow\ T \end{array}$$

proves

$$(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![\Gamma \vdash T]\!]\,\rho \tag{5.11}$$

We therefore obtain

(S3) $v \in \|A[\rho/\Gamma]\|_{\Sigma'}$, by *(C3)* and (5.10)

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![\Gamma \vdash T]\!]\,\rho$, by (5.11)

as required.

· Case (AL OBJ)
Suppose *(H1)*: $\Gamma \vdash a : A :: T$ has been derived by an application of rule the (AL OBJ) rule. Necessarily $a \equiv [\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J}$. Suppose that

(H2) Σ is a store specification

(H3) $\rho \in \|\Gamma\|_\Sigma$

We recall the object introduction rule,

$$\text{(AL OBJ)} \qquad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\,A_i,\ \mathsf{m}_j{:}\,\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x_i{:}A_i :: T_{\mathsf{res}}(x_i)^{\,i=1\ldots n} \quad \Gamma, y_j{:}A \vdash b_j{:}B_j{::}T_j{}^{\,j=1\ldots m} \end{array}}{\Gamma \vdash \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)b_j\right]_{i\in I, j\in J}{:}A{::}T_{\mathsf{obj}}(\mathsf{f}_i = x_i)_{i\in I}}$$

from which we see that A is $[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j{::}T_j]_{i\in I, j\in J}$, that T is $T_{\mathsf{obj}}(\mathsf{f}_i = x_i)_{i\in I}$ and that

(H1') $\Gamma \vdash x_i : A_i :: T_{\mathsf{res}}(x_i)$ for $i \in I$

(H1") $\Gamma, y_j{:}A \vdash b_j : B_j :: T_j$ for $j \in J$

We have to show that there is $\phi \in \mathsf{SF}$ such that for all $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$ and $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ with $[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle$, conditions *(S1)–(S4)* hold.

From *(H3)* and Lemma 5.1.6 we know that for all $\hat{\Sigma} \succcurlyeq \Sigma$ and $l_0 \in \|A[\rho/\Gamma]\|_{\hat{\Sigma}}$,

$$\rho[y_j := l_0] \in \left|\left|\Gamma, y_j{:}A\right|\right|_{\hat{\Sigma}}$$

Hence by induction hypothesis on *(H1")*, for all $j \in J$ there is $\phi^j_{l_0} \in \mathsf{SF}$ such that for all $\Sigma_1 \succcurlyeq \hat{\Sigma}$, for all $l_1 \in \mathsf{Loc}$ and for all $\langle \sigma_1, \phi_1 \rangle \in \mathit{fix}(\Phi)_{\hat{\Sigma}}$ with $[\![b_j]\!]\,\rho[y_j := l_0]\sigma_1 = \langle v_2, \sigma_2 \rangle$, we obtain the conclusions *(S1)–(S4)* of the lemma, i.e.,

(S1') there exists $\Sigma_2 \succcurlyeq \Sigma_1$ and $\phi_2 \in \mathsf{RSF}$ such that $\phi^j_{l_0}(l_1, \sigma_1, \phi_1, \Sigma_1) = \langle \Sigma_2, \phi_2 \rangle$

(S2') $\langle \sigma_2, \phi_2 \rangle \in \mathit{fix}(\Phi)_{\Sigma_2}$

(S3') $v_2 \in \left|\left|B_j[\rho[y_j := l_0]/\Gamma, y_j{:}A]\right|\right|_{\Sigma_2}$

(S4') $(\pi_{\mathrm{Val}}(\sigma_1), v_2, \pi_{\mathrm{Val}}(\sigma_2)) \in [\![[\Gamma, y_j{:}A] \vdash T_j]\!]\,\rho[y_j := l_0]$

For any location l we have $r_l \stackrel{def}{=} \{\!|\, l = \{\!|\, \mathsf{m}_j = \lambda(l_0, \sigma_0, \phi_0, \Sigma_0).\phi^j_{l_0}(l_0, \sigma_0, \phi_0, \Sigma_0) \,|\!\}_{j \in J} \,|\!\} \in \mathsf{RSF}$, therefore we can define $\phi \in \mathsf{SF}$ by

$$\phi(l, \sigma', \phi', \Sigma') \stackrel{def}{=} \begin{cases} \langle \Sigma' + \{\!|\, l_0 = A[\rho/\Gamma] \,|\!\}, \phi' + r_{l_0} \rangle \\ \qquad\qquad \text{if } \Sigma' \succcurlyeq \Sigma, l_0 \notin \mathsf{dom}(\Sigma') \text{ and } [\![a]\!]\,\rho\sigma' = \langle l_0, \sigma'' \rangle \\ \text{undefined} \quad \text{otherwise} \end{cases} \tag{5.12}$$

We show that *(S1)–(S4)* hold. Let $\Sigma' \succcurlyeq \Sigma$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\hat{\Sigma}}$ and suppose $[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle$. By definition of the semantics, and the fact that *(H1')* entails $\rho(x_i)\downarrow$ for all $i \in I$, for

$$[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle \in \mathsf{Loc} \times \mathsf{St} \tag{5.13}$$

we obtain $v = l_0$ where $l_0 \notin \mathsf{dom}(\sigma')$ (and so $l_0 \notin \mathsf{dom}(\Sigma')$) and

$$\sigma'' \;=\; \sigma' + \{\!|\, l_0 = \{\!|\, \mathsf{f}_i = \rho(x_i), \mathsf{m}_j = \lambda\langle l, \sigma \rangle.\, [\![b_j]\!]\,\rho[y_j := l]\sigma \,|\!\}_{i \in I, j \in J} \,|\!\} \tag{5.14}$$

We obtain that there exists $\phi'' \in \mathsf{RSF}$ s.t.

(S1) $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma' + \{\!|\, l_0 = A[\rho/\Gamma] \,|\!\}, \phi'' \rangle$, by construction of ϕ in equation (5.12)

(S3) $v = l_0 \in \|A[\rho/\Gamma]\|_{\Sigma', l_0:A[\rho/\Gamma]}$, by definition of $\|\cdot\|$

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![\Gamma \vdash T_{\mathsf{obj}}(\mathsf{f}_i = x_i)_{i \in I}]\!]\,\rho$, which is easily checked from the definition of $T_{\mathsf{obj}}(\mathsf{f}_i = x_i)_{i \in I}$, the semantics in Table 4.6 and equation (5.14).

All that remains to be shown is *(S2)*: $\langle \sigma'', \phi'' \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$, where Σ'' is $\Sigma' + \{\!|\, l_0 = A[\rho/\Gamma] \,|\!\}$. By the construction of ϕ in (5.12),

$$\phi'' \;=\; \phi' + \{\!|\, l_0 = \{\!|\, \mathsf{m}_j = \phi^j_{l_0} \,|\!\}_{j \in J} \,|\!\} \tag{5.15}$$

and we show that *(S2)* holds according to Definition 5.1.7 next:

As for the first condition, (Dom), by assumption the domains of ϕ' and Σ' agree, and by construction of ϕ, also $\mathsf{dom}(\phi''.l_0) = \{\mathsf{m}_j \mid j \in J\} = \mathsf{dom}(\pi_{mthds}(\Sigma''.l_0))$. To check the second condition, suppose $l \in \mathsf{dom}(\Sigma'')$. We distinguish two cases: For locations $l \neq l_0$ the conditions follow easily from the assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ about the initial store σ'. For the freshly allocated object at location l_0 the previous considerations lead to the result.

– $l \neq l_0$: Then

$$\Sigma''.l \;=\; \Sigma'.l \;=\; [\mathsf{g}_i{:}A'_i, \mathsf{n}_j{:}\varsigma(y_j)B'_j :: T'_j]_{i\in I', j\in J'}$$

(F) For all $i \in I'$, $\sigma''.l.\mathsf{g}_i = \sigma'.l.\mathsf{g}_i$, and so from $\langle\sigma', \phi'\rangle \in \mathit{fix}(\Phi)_{\Sigma'}$

$$\sigma''.l.\mathsf{g}_i \in ||A'_i||_{\Sigma'} \subseteq ||A'_i||_{\Sigma''}$$

(M) Let $j \in J'$, let $\Sigma_1 \succcurlyeq \Sigma''$, let $l' \in \|\Sigma''.l\|_{\Sigma_1}$, let $\langle\sigma_1, \phi_1\rangle \in \mathit{fix}(\Phi)_{\Sigma_1}$ and suppose $\sigma''.l.\mathsf{n}_j(\sigma_1) = \langle\nu_2, \sigma_2\rangle$. Since $\sigma''.l.\mathsf{n}_j = \sigma'.l.\mathsf{n}_j$ and $\Sigma_1 \succcurlyeq \Sigma'$, the assumption $\langle\sigma', \phi'\rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and the construction of ϕ'' yield

* $\phi''.l.\mathsf{n}_j(l', \sigma_1, \phi_1, \Sigma_1) = \phi'.l.\mathsf{n}_j(l', \sigma_1, \phi_1, \Sigma_1) = \langle\Sigma_2, \phi_2\rangle$
* $\langle\sigma_2, \phi_2\rangle \in \mathit{fix}(\Phi)_{\Sigma_2}$
* $\nu_2 \in \left|\left|B'_j[l'/y_j]\right|\right|_{\Sigma_2}$
* $(\pi_{\mathrm{Val}}(\sigma_1), \nu_2, \pi_{\mathrm{Val}}(\sigma_2)) \in [\![T'_j[l'/y_j]]\!]$

– $l = l_0$:

(F) By assumption *(H1')* and $\rho \in \|\Gamma\|_\Sigma$ we know that for all $i \in I$ there are A'_i such that $\vdash A_i{}' \prec: A_i$ and $x_i{:}A_i{}'$ in Γ. Hence,

$$\sigma''.l_0.\mathsf{f}_i = \rho(x_i) \in ||A_i{}'||_\Sigma \subseteq \|A_i\|_\Sigma \subseteq \|A_i\|_{\Sigma''}$$

(M) Let $j \in J$. Suppose $\Sigma_1 \succcurlyeq \Sigma''$, let $l' \in \|\Sigma''.l_0\|_{\Sigma_1}$, let $\langle\sigma_1, \phi_1\rangle \in \mathit{fix}(\Phi)_{\Sigma_1}$ and suppose

$$\sigma''.l_0.\mathsf{m}_j(l', \sigma_1) \;=\; \langle\nu_2, \sigma_2\rangle$$

Since $\sigma''.l_0.\mathsf{m}_j(l', \sigma_1) = [\![b_j]\!]\,\rho[y_j := l']\sigma_1$ and $\Sigma_1 \succcurlyeq \Sigma'' \succcurlyeq \Sigma$, the assumption $\langle\sigma_1, \phi_1\rangle \in \mathit{fix}(\Phi)_{\Sigma_1}$ and the construction of ϕ'' yield $\Sigma_2 \succcurlyeq \Sigma_1$ and ϕ_2 such that

* $\phi''.l_0.\mathsf{m}_j(l', \sigma_1, \phi_1, \Sigma_1) = \phi^j_{l'}(l', \sigma_1, \phi_1, \Sigma_1) = \langle\Sigma_2, \phi_2\rangle$, by *(S1')*
* $\langle\sigma_2, \phi_2\rangle \in \mathit{fix}(\Phi)_{\Sigma_2}$, by *(S2')*
* $\nu_2 \in \left|\left|B_j[\rho[y_j := l']/\Gamma, y_j{:}A]\right|\right|_{\Sigma_2} = \left|\left|B_j[\rho/\Gamma][l'/y_j]\right|\right|_{\Sigma_2}$, by *(S3')*
* $(\pi_{\mathrm{Val}}(\sigma_1), \nu_2, \pi_{\mathrm{Val}}(\sigma_2)) \in [\![[\Gamma, y_j{:}A] \vdash T_j]\!]\,\rho[y_j := l'] = [\![T_j[\rho/\Gamma][l'/y_j]]\!]$, by *(S4')*

where the equations in the last two lines are by the substitution lemma, Lemma 5.1.2.

Thus we have shown $\langle\sigma'', \phi''\rangle \in \mathit{fix}(\Phi)_{\Sigma''}$, i.e., *(S2)* holds.

· Case (AL Inv)
Suppose $\Gamma \vdash a : A :: T$ is derived by an application of the method invocation rule:

$$\text{(AL Inv)} \qquad \frac{\Gamma \vdash x{:}[\mathsf{m}{:}\varsigma(y)A' :: T']::T_{\mathsf{res}}(x)}{\Gamma \vdash x.\mathsf{m}{:}A'[x/y] :: T'[x/y]}$$

Necessarily a is of the form $x.\mathsf{m}$ and there are A' and T' s.t. $A \equiv A'[x/y]$ and $T \equiv T'[x/y]$. So suppose

(H1) $\Gamma \vdash a : A'[x/y] :: T'[x/y]$

(H2) $\Sigma \in \mathit{StSpec}$ is a store specification

(H3) $\rho \in \|\Gamma\|_\Sigma$

Define $\phi \in \mathsf{SF}$ using "self-application" of the argument ϕ',

$$\phi(l, \sigma', \phi', \Sigma') \quad \stackrel{def}{=} \quad \phi'.\rho(x).\mathsf{m}(\rho(x), \sigma', \phi', \Sigma') \tag{5.16}$$

Now let $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and suppose $[\![a]\!]\,\rho\sigma' = \sigma'.\rho(x).\mathsf{m}(\sigma') = \langle v, \sigma'' \rangle$ terminates. We show that *(S1)–(S4)* hold.

By the hypothesis of the method invocation rule,

$$\Gamma \vdash x{:}[\mathsf{m}{:}\varsigma(y)A' :: T'] :: T_{\mathsf{res}}(x) \tag{H1'}$$

Since this implies $x{:}B \in \Gamma$ for some $[\Gamma] \vdash B \prec: [\mathsf{m} : \varsigma(y)A' :: T']$, by assumption *(H3)* and the definition of $\Sigma' \succcurlyeq \Sigma$ this entails

$$\vdash \Sigma'.(\rho(x)) \prec: [\mathsf{m} : \varsigma(y)A' :: T']\,[\rho/\Gamma]$$

i.e., there are I, J and for all $i \in I$ and $j \in J$, there exist A_i, A'', B_j and T_j, T'' such that

$$\Sigma'.\rho(x) \quad \equiv \quad [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B_j :: T_j, \mathsf{m}{:}\varsigma(y)A'' :: T'']_{i \in I, j \in J}$$

where

$$y \vdash A'' \prec: A'[\rho/\Gamma] \quad \text{and} \quad \vdash_{\mathsf{fo}} T'' \to T'[\rho/\Gamma] \tag{5.17}$$

Note that $\rho(x) \in \|\Sigma'.\rho(x)\|_{\Sigma'}$. Thus assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ with equation (5.16) implies that there are Σ'', ϕ'' such that $\Sigma'' \succcurlyeq \Sigma'$ and

(S1) $\phi(l, \sigma', \phi', \Sigma') = \phi'.(\rho(x)).\mathsf{m}(\rho(x), \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$

(S2) $\langle \sigma'', \phi'' \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$

(S3') $v \in \|A''[\rho(x)/y]\|_{\Sigma''}$

(S4') $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![\vdash T''[\rho(x)/y]]\!]$

By transitivity of $\prec:$ equation (5.17), Lemma 5.1.2 and *(S3')* yield

$$v \in \big|\big|A'[\rho/\Gamma][\rho(x)/y]\big|\big|_{\Sigma''}$$

Since $A'[\rho/\Gamma, \rho(x)/y] \equiv A'[x/y][\rho/\Gamma]$ we also have

(S3) $v \in \|A'[x/y][\rho/\Gamma]\|_{\Sigma''} = \|A[\rho/\Gamma]\|_{\Sigma''}$

Similarly, by (5.17) and *(S4')*,

$$\begin{aligned}(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T''[\rho(x)/y]]\!] &\subseteq [\![T'[\rho/\Gamma][\rho(x)/y]]\!] \\ &= [\![[\Gamma] \vdash T[x/y]]\!]\,\rho \end{aligned} \tag{S4}$$

which was to show.

- Case (AL SEL)
  Similar to the previous case. The choice function ϕ can be defined as $\phi(l, \sigma', \phi', \Sigma') \stackrel{def}{=} \langle \phi', \Sigma' \rangle$, reflecting the fact the store does not change.

- Case (AL UPD)
  Suppose

(H1) $\Gamma \vdash a{:}A{::}T$ has been derived by an application of the rule (AL UPD),

(H2) Σ is a store specification

(H3) $\rho \in \|\Gamma\|_\Sigma$

Define $\phi \in \mathsf{SF}$ by $\phi(l, \sigma', \phi', \Sigma') \stackrel{def}{=} \langle \Sigma', \phi' \rangle$. This reflects the fact that store specifications are necessarily preserved when updating fields.

Let $\Sigma' \succcurlyeq \Sigma$, $l \in \mathsf{Loc}$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and suppose $[\![a]\!]\,\rho\sigma' = \langle v, \sigma'' \rangle$ terminates. Recall the rule for field update,

$$\text{(AL UPD)} \qquad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma \vdash x{:}A{::}T_{\mathsf{res}}(x) \qquad \Gamma \vdash y{:}A_k{::}T_{\mathsf{res}}(y) \qquad k \in I \end{array}}{\Gamma \vdash x.\mathsf{f}_k := y{:}A{::}T_{\mathsf{upd}}(x, \mathsf{f}_k, y)}$$

In particular, a is of the form $x.\mathsf{f}_k := y$ and T is $T_{\mathsf{upd}}(x, \mathsf{f}_k, y)$. From the semantics of $[\![a]\!]\,\rho\sigma'$, this means $v = \rho(x) \in \mathsf{Loc}$ and

$$\sigma'' \;=\; \sigma'[v := \sigma'.v[\mathsf{f}_k := \rho(y)]] \tag{5.18}$$

We show that *(S1)–(S4)* hold.

By *(H3)*, $\rho(x) \in \|A[\rho/\Gamma]\|_\Sigma \subseteq \|A[\rho/\Gamma]\|_{\Sigma'}$. Then by construction of ϕ, and (5.18),

(S1) $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma', \phi' \rangle$; in particular, $\Sigma' \succcurlyeq \Sigma'$

(S3) $v = \rho(x) \in \|A[\rho/\Gamma]\|_{\Sigma'}$

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T]\!]\,\rho$, from the semantics given in Table 4.6

It remains to show *(S2)*, $\langle \sigma'', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, which we prove next.

By assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, the first condition of Definition 5.1.7, requiring the domains of ϕ' and Σ' to agree, is clearly satisfied. As for the further conditions, suppose $l \in \mathsf{dom}(\Sigma')$ such that

$$\Sigma'.l \quad \equiv \quad [\mathsf{g}_i{:}A'_i,\ \mathsf{n}_j{:}\varsigma(y_j)B'_j :: T'_j]_{i\in I', j\in J'}$$

(F) We distinguish two cases:

- Case $l = \rho(x)$ and $\mathsf{g}_i = \mathsf{f}_k$. Then, by (5.18), $\sigma''.l.\mathsf{g}_i = \rho(y)$. By *(H3)*, $\rho(x) \in \|A[\rho/\Gamma]\|_\Sigma \subseteq \|A[\rho/\Gamma]\|_{\Sigma'}$, which entails

  $$\vdash \Sigma'.l \prec: A[\rho/\Gamma]$$

  and in particular, by the definition of the subspecification relation, $A'_k \equiv A_k[\rho/\Gamma]$. Note that *invariance of subspecification* in the field components is needed to conclude this. Now again by *(H3)*,

  $$\rho(y) \in \|A_k[\rho/\Gamma]\|_\Sigma \subseteq \|A_k[\rho/\Gamma]\|_{\Sigma'} = \left\|A'_k\right\|_{\Sigma'}$$

  Hence, $\sigma''.l.\mathsf{g}_i \in \left\|A'_i\right\|_{\Sigma'}$ as required.
- Case $l \neq \rho(x)$ or $\mathsf{g}_i \neq \mathsf{f}_k$. Then $\sigma''.l.\mathsf{g}_i = \sigma'.l.\mathsf{g}_i$, by (5.18). Hence, by assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, we have $\sigma''.l.\mathsf{g}_i \in \left\|A'_i\right\|_{\Sigma'}$.

(M) Let $\Sigma'' \succcurlyeq \Sigma'$, let $l' \in \|\Sigma'.l\|_{\Sigma''}$, let $\langle \sigma_1, \phi_1 \rangle \in \mathit{fix}(\Phi)_{\Sigma''}$ and suppose $\sigma''.l.\mathsf{n}_j(\sigma_1) = \langle v_2, \sigma_2 \rangle$. Then, by assumption $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$ and the fact that $\sigma''.l.\mathsf{n}_j = \sigma'.l.\mathsf{n}_j$ by (5.18), we obtain that $\phi'.l.\mathsf{n}_j(l', \sigma_1, \phi_1, \Sigma'') = \langle \Sigma_2, \phi_2 \rangle$ s.t. $\Sigma_2 \succcurlyeq \Sigma''$ and

(M1) $(\pi_{\mathrm{Val}}(\sigma_1), \nu_2, \pi_{\mathrm{Val}}(\sigma_2)) \in [\![T'_j[l'/y_j]]\!]$
(M2) $\langle \sigma_2, \phi_2 \rangle \in \mathit{fix}(\Phi)_{\Sigma_2}$
(M3) $\nu_2 \in \left|\!\left|B'_j[l'/y_j]\right|\!\right|_{\Sigma_2}$

as required.

· Case (AL COPY)
Suppose

(H1) $\Gamma \vdash a{:}A{::}T$ has been derived by an application of the rule (AL COPY),

(H2) Σ is a store specification

(H3) $\rho \in \|\Gamma\|_\Sigma$

Thus, by inspection of the rule (AL COPY), $A \equiv [\mathsf{f}_i{:}\, A_i,\, \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ and $T \equiv \exists \overline{y}.\ \bigwedge_{i\in I} y_i = \mathsf{sel}_{pre}(x, \mathsf{f}_i)\ \wedge\ T_{\mathsf{obj}}(\mathsf{f}_i = y_i)_{i\in I}$ and a is of the form $\mathsf{clone}\, x$. Now define $\phi \in \mathsf{SF}$ by $\phi(l, \sigma', \phi', \Sigma') \stackrel{def}{=} \langle \Sigma'', \phi'' \rangle$ where

$$\Sigma'' \stackrel{def}{=} \Sigma' + \{\!|\, l_0 = A[\rho/\Gamma] \,|\!\} \quad \text{where } l_0 \notin \mathsf{dom}(\Sigma')\ \wedge\ l_0 \notin \mathsf{dom}(\phi')$$

$$\phi'' \stackrel{def}{=} \begin{cases} \phi' + \{\!|\, l_0 = \{\!|\, \mathsf{m}_j = \phi'.(\rho(x)).\mathsf{m}_j \,|\!\}_{j\in J} \,|\!\} & \text{if } \phi'.(\rho(x)).\mathsf{m}_j \downarrow \quad \forall\, j \in J \\ \text{undefined} & \text{otherwise} \end{cases}$$

Suppose $\Sigma' \succcurlyeq \Sigma$, $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, $l \in \mathsf{Loc}$ and there are $v \in \mathsf{Val}$ and $\sigma'' \in \mathsf{St}$ such that

$$[\![a]\!]\, \rho\sigma' \ = \ \langle v, \sigma'' \rangle$$

By definition of ϕ above,

(S1) $\phi(l, \sigma', \phi', \Sigma') = \langle \Sigma'', \phi'' \rangle$ for some $\Sigma'' \succcurlyeq \Sigma'$ and ϕ''

Furthermore, from the denotational semantics of cloning,

$$\langle v, \sigma'' \rangle \ = \ [\![a]\!]\, \rho\sigma' \ = \ \langle l_0, \sigma'[l_0 := \sigma'.(\rho(x))] \rangle$$

Thus, both

(S3) $v \in [\![A[\rho/\Gamma]]\!]_{\Sigma''}$

(S4) $(\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![[\Gamma] \vdash T]\!]\, \rho$

follow, from the definition of ϕ above and the definition of $[\![[\Gamma] \vdash T]\!]$, resp. It remains to prove condition *(S2)*. This follows much as in the case (AL UPD) exploiting the fact that all locations distinct from l_0 remain unchanged, and that moreover both the object stored at l_0 and its specification $\Sigma''.l_0$ are the same as the ones at location $\rho(x)$ in the original store and store specification, resp.

This concludes the proof. □

5.2.3 Soundness Theorem

With Lemma 5.2.1 and Lemma 5.2.5, proved in Sections 5.2.1 and 5.2.2, it is now easy to establish our main result:

Theorem 5.2.6 (Soundness)**.** *If* $\Gamma \vdash a : A :: T$ *then* $\Gamma \vDash a : A :: T$.

Proof. Suppose $\Gamma \vdash a : A :: T$, and let $\Sigma \in \mathit{StSpec}$ be a store specification and suppose $\rho \in \mathsf{Env}$ such that $\rho \in \|\Gamma\|_\Sigma$. Let $\sigma \in [\![\Sigma]\!]$, so by definition there exists $\phi \in \mathsf{RSF}$ such that $\langle \sigma, \phi \rangle \in \mathit{fix}(\Phi)_\Sigma$. Next suppose

$$[\![a]\!]\,\rho\sigma = \langle v, \sigma' \rangle$$

By Lemma 5.2.5 there exists $\phi_a \in \mathsf{SF}$ such that for all $l \in \mathsf{Loc}$,

$$\phi_a(l, \sigma, \phi, \Sigma) = \langle \Sigma', \phi' \rangle$$

where $\Sigma' \succcurlyeq \Sigma$ and $\langle \sigma', \phi' \rangle \in \mathit{fix}(\Phi)_{\Sigma'}$, i.e., $\sigma' \in [\![\Sigma']\!]$ follows. Moreover,

- $v \in \|A[\rho/\Gamma]\|_{\Sigma'}$, and
- $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T]\!]\,\rho$

In the case where A is bool we now obtain $\langle v, \sigma' \rangle \in [\![[\Gamma] \vdash A]\!]\,\rho$ from $\|\mathsf{bool}\|_{\Sigma'} = \mathsf{BVal}$. Otherwise A is an object specification, and we must have $\vdash \Sigma'.v \prec: A[\rho/\Gamma]$ by definition of $\|A[\rho/\Gamma]\|_{\Sigma'}$. Hence, by Lemma 5.2.1,

$$\langle v, \sigma' \rangle \in [\![A[\rho/\Gamma]]\!] = [\![[\Gamma] \vdash A]\!]\,\rho$$

where the last equality is by the the substitution lemma, Lemma 5.1.2. □

For programs we also obtain the following corollaries to the soundness theorem.

Corollary 5.2.7 (Closed Terms)**.** *If* $\emptyset \vdash a : A :: T$ *and* $[\![a]\!]\,\sigma = \langle v, \sigma' \rangle$ *then* $\langle v, \sigma' \rangle \in [\![A]\!]$. *In particular,* $v \neq \mathsf{error}$ *by Lemma 4.3.3.*

Corollary 5.2.8 (Type Soundness)**.** *For (first-order) types A, if* $\emptyset \rhd a : A$ *and* $[\![a]\!]\,\sigma = \langle v, \sigma' \rangle$ *then* $v \neq \mathsf{error}$.

Corollary 5.2.8 follows from the observation that $\Gamma \rhd a : A$ is derivable if and only if $\Gamma^* \vdash a : A^* :: \mathit{True}$ is derivable, where A^* is the specification obtained from the type A by augmenting every nested method type with the trivial transition relation *True*, see (Abadi and Leino 2004).

5.2.4 Comparison to Object Encodings

It is interesting to compare the semantics of the object calculus with the object encodings discussed in Chapter 3. As described above, types can be seen as a particular class of specifications of Abadi and Leino's logic, and Theorem 5.2.6 entails a type soundness result with respect to this induced semantics of types.

Suppose $A \equiv [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$. Firstly recall that there are in fact two interpretations of types A, as result specifications $\|A\|$ and $[\![A]\!]$, respectively, the first of which are defined with respect to a syntactic store specification and abstract away from concrete stores, while the latter denote predicates over the domain $\mathsf{Val} \times \mathsf{St}$. However, this latter interpretation only provides a very partial description of program properties: $[\![A]\!]$ is defined in terms of interaction with the object through field lookup and method invocation only, but *not* using field update.

In contrast, result specifications $\|A\|_\Sigma$ are defined in terms of the store specification Σ: The property $l \in \|A\|_\Sigma$ holds if l is a location, specified as A in Σ (possibly using the subspecification relation $\prec:$ which is defined inductively from the inference rule (SUBSPEC OBJ). Thus, the "proper" meaning of $l \in \|A\|_\Sigma$ is obtained through the definition of store specifications, $\sigma \in [\![\Sigma]\!]$, linking the location l to an actual object, $\sigma.l$, in the store. In particular, we find that an object at such a location l has to provide methods m_j that, when the self parameter is applied to the location of any other object satisfying at least (with respect to $\prec:$) A, yield results of type B_j. So the use of store specifications serves, amongst other things, the purpose of expressing a type recursion, similar to the imperative self application semantics of (Abadi, Cardelli, and Viswanathan 1996). Recalling the definition of store specifications, our semantics corresponds roughly to a type

$$\mu(Y)\forall(X \prec: Y).\,\{\mathsf{f}_i : A_i,\, \mathsf{m}_j : X \Rightarrow B_j\}_{i,j} \tag{5.19}$$

rather than their encoding

$$\mu(Y)\exists(X \prec: Y).\,\{\mathsf{self} : X,\, \mathsf{f}_i : A_i,\, \mathsf{m}_j : X \Rightarrow B_j\}_{i,j} \tag{5.20}$$

While the bounded existential in (5.20) allows for the subtyping of the contravariant occurrence of self, the bounded universal (5.19) clearly does not. This provides a more abstract explanation of why subtyping of store specifications (in the sense of pointwise subtyping in each location) must fail.

We note that it would be interesting to obtain a semantics of object types like (5.20) also in the case of imperative objects. However, we are not aware of any work on semantics of higher-order store and existential types.

Chapter 6

Recursive Specifications

In this section we investigate an extension of Abadi and Leino's logic with recursive specifications. In analogy with recursive types, these are necessary when a field of an object or the result of one of the object's methods are supposed to satisfy the same specification as the object itself. In particular, they are needed to specify any recursive datatype: Referring back to the example of the account manager in Table 4.1, if $A_{\mathtt{Manager}}$ should include a list of accounts, we would need a recursive specification A such that

$$\langle l, \sigma\rangle \in [\![A]\!] \quad \Longleftrightarrow \quad \langle \sigma.l.\mathsf{head}, \sigma\rangle \in [\![A_{\mathtt{Account}}]\!] \ \wedge \ \langle \sigma.l.\mathsf{tail}, \sigma\rangle \in [\![A]\!] \tag{6.1}$$

We extend the syntax of specifications, so that the recursive specification

$$\mu X.\ [\mathsf{head} : A_{\mathtt{Account}}, \mathsf{tail} : X] \tag{6.2}$$

denotes the predicate (6.1).

Below we discuss in more detail how recursive specifications can be dealt with in the logic. Section 6.1 presents the syntax of recursive specifications along with a number of proof rules. The semantics of recursive specifications is given in Section 6.2; analoguously to the preceding chapter we again define a semantics of store specifications. Section 6.3 adapts the soundness proof: The main difficulty is in establishing the analogue of Lemma 5.2.1; see Section 6.3.1.

6.1 Syntax and Proof Rules

The introduction of recursive specifications $\mu(X)A$ proceeds very much like the development of recursive *types* of Chapter 3: Meaningless specifications such as $\mu(X)X$ are prevented by only allowing recursion through object specifications.

$$\begin{array}{ll} \underline{A}, \underline{B} & ::= \top \ \mid \ \mathsf{bool} \ \mid \ [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \ \mid \ \mu(X)\underline{A} \\ A, B \in \mathit{RecSpec} & ::= \underline{A} \ \mid \ X \end{array} \tag{6.3}$$

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

Table 6.1 *Well-formed recursive object specifications and contexts*

(CONTEXT EMP) $$\frac{}{\emptyset;\emptyset \vdash \mathsf{ok}}$$

(CONTEXT SPEC) $$\frac{\Gamma;\Delta \vdash \mathsf{ok} \qquad \Gamma;\Delta \vdash A \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x{:}A;\Delta \vdash \mathsf{ok}}$$

(CONTEXT VAR) $$\frac{\Gamma;\Delta \vdash Y \qquad X \notin \Gamma}{\Gamma;\Delta, X \prec: Y \vdash \mathsf{ok}}$$

(CONTEXT TOP) $$\frac{\Gamma;\Delta \vdash \mathsf{ok} \qquad X \notin \Gamma}{\Gamma;\Delta, X \prec: \top \vdash \mathsf{ok}}$$

(SPEC VAR) $$\frac{\Gamma;\Delta, X \prec: A, \Delta' \vdash \mathsf{ok}}{\Gamma;\Delta, X \prec: A, \Delta' \vdash X}$$

(SPEC CONST) $$\frac{\Gamma;\Delta \vdash \mathsf{ok}}{\Gamma;\Delta \vdash \top} \qquad \frac{\Gamma;\Delta \vdash \mathsf{ok}}{\Gamma;\Delta \vdash \mathsf{bool}}$$

(SPEC OBJ) $$\frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \\ \Gamma;\Delta \vdash A_i \quad \forall i \in I \qquad \Gamma, y_j{:}A;\Delta \vdash B_j \qquad [\Gamma, y_j{:}A] \vdash T_j \quad \forall j \in J \end{array}}{\Gamma;\Delta \vdash [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}}$$

(SPEC REC) $$\frac{\Gamma;\Delta, X \prec: \top \vdash A}{\Gamma;\Delta \vdash \mu(X)A}$$

We introduce contexts Δ that contain specification variables with an upper bound, $X \prec: A$, where A is either another variable or $\top$. These contexts play the same rôle as contexts for recursive types (cf. Section 3.3.1). However, specifications may also contain free *term* variables. Thus we additionally need the specification contexts Γ of Section 4.3.1: In the rules of the logic we replace every judgement $\Gamma \vdash a : A :: T$ by $\Gamma;\Delta \vdash a : A :: T$, and the definitions of well-formed specifications and well-formed specification contexts are extended. The inference rules defining these notions are contained in Table 6.1. We often write Δ, X, Δ' for $\Delta, X \prec: \top, \Delta'$.

Subspecifications for recursive specifications are obtained by analogy to the "usual" recursive subtyping rule (Amadio and Cardelli 1993): The subspecification relation $\Gamma;\Delta \vdash A \prec: B$ is induced by

(SUBSPEC REC) $$\frac{\Gamma;\Delta, Y \prec: \top, X \prec: Y \vdash A \prec: B}{\Gamma;\Delta \vdash \mu X.A \prec: \mu Y.B}$$

In particular, compare this to the rule (RECSUB REC) of Table 3.5 on page 45. $\top$ is the greatest specification. More precisely, the subspecification relation $\Gamma;\Delta \vdash A \prec: B$ is the least reflexive transitive relation closed under the rules summarised in Table 6.2.

In the rule (SUBSPEC OBJ) we use the trivial specification $\top$ for the self parameter y_j in the antecedent $\Gamma, y_j{:}\top;\Delta \vdash B_j$ that a method result specification B_j is well-formed, and similarly in the antecedent $\Gamma, y_j{:}\top;\Delta \vdash B_j \prec: B'_j$ that indeed $B_j \prec: B'_j$ holds. Note that this works since well-formedness essentially depends on verifying that the nested transition relations do not have free variables other than the ones appearing in the context. Since the assertion language for transition relations is untyped first-order logic, the precise types of term variables are not relevant for this purpose. In fact, there wouldn't be a type for

Table 6.2 *Subspecification relation for recursive object specifications*

(SUBSPEC TOP)
$$\frac{\Gamma;\Delta \vdash A}{\Gamma;\Delta \vdash A \prec: \top}$$

(SUBSPEC OBJ)
$$\frac{\begin{array}{llll} \Gamma;\Delta \vdash A_i & \forall i \in I & \Gamma, y_j{:}\top;\Delta \vdash B_j & \forall j \in J \\ [\Gamma, y_j{:}\top] \vdash T_j & \forall j \in J & [\Gamma, y_j{:}\top] \vdash T'_j & \forall j \in J' \quad I' \subseteq I \\ \Gamma, y_j{:}\top;\Delta \vdash B_j \prec: B'_j & \forall j \in J' & \vdash_{\mathsf{fo}} T_j \to T'_j & \forall j \in J' \quad J' \subseteq J \end{array}}{\Gamma;\Delta \vdash [\mathsf{f}_i{:}A_i,\, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \prec: [\mathsf{f}_i{:}A_i,\, \mathsf{m}_j{:}\varsigma(y_j)B'_j{::}T'_j]_{i\in I', j\in J'}}$$

(SUBSPEC REC)
$$\frac{\Gamma;\Delta, Y \prec: \top, X \prec: Y \vdash A \prec: B}{\Gamma;\Delta \vdash \mu X.A \prec: \mu Y.B}$$

(SUBSPEC VAR)
$$\frac{\Gamma;\Delta, X \prec: A, \Delta' \vdash \mathsf{ok}}{\Gamma;\Delta, X \prec: A, \Delta' \vdash X \prec: A}$$

(SUBSPEC FOLD)
$$\frac{\Gamma;\Delta \vdash \mu X.A}{\Gamma;\Delta \vdash A[(\mu X.A)/X] \prec: \mu X.A}$$

(SUBSPEC UNFOLD)
$$\frac{\Gamma;\Delta \vdash \mu X.A}{\Gamma;\Delta \vdash \mu X.A \prec: A[(\mu X.A)/X]}$$

the "auxiliary" variables ranging over field names and locations (see also the discussion following Remark 4.3.2 on page 75).

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., $[\![\mu X.A]\!] = [\![A[(\mu X.A)/X]]\!]$. Because of this, we do not need to introduce *fold* and *unfold* terms: We can deal with folding and unfolding of recursive specifications through the subsumption rule once we close the subspecification relation under the rules (SUBSPEC FOLD) and (SUBSPEC UNFOLD). We will prove their soundness below.

First, let us consider a simple example. Let $a \equiv [\mathsf{f} = 0,\, \mathsf{m} = \varsigma(y)\, y.\mathsf{f}{:=}y.\mathsf{f} + 1;\, y]$ be an object with a method returning self, after incrementing its field f. A specification of its behaviour is the following

$$A \;\equiv\; \mu(X)\,[\mathsf{f} : \mathsf{int},\, \mathsf{m} : \varsigma(y)X{::}\mathsf{result} = y \;\wedge\; \mathsf{sel}_{post}(y,\mathsf{f}) = \mathsf{sel}_{pre}(y,\mathsf{f}) + 1]$$

we abbreviate the transition relation by T. The proof of $\vdash a : A$ is simple: By (AL SUB) and (AL OBJ) we obtain

$$\frac{\dfrac{\vdash 0 : \mathsf{int}{::}T_{\mathsf{res}}(0) \qquad y{:}A \vdash y.\mathsf{f}{:=}y.\mathsf{f} + 1;\, y : A{::}T}{\vdash a : [\mathsf{f} : \mathsf{int},\, \mathsf{m} : \varsigma(y)A{::}T] :: T_{\mathsf{obj}}(\mathsf{f} = 0)}}{\vdash a : A :: T_{\mathsf{obj}}(\mathsf{f} = 0)}$$

where $\vdash [\mathsf{f} : \mathsf{int},\, \mathsf{m} : \varsigma(y)A{::}T] \prec: A$ holds by (SUBSPEC FOLD). The proof of $\vdash 0 : \mathsf{int}{::}T_{\mathsf{res}}(0)$ is trivial, the proof of the second obligation proceeds as in the system without recursive specifications: By (AL UPD) and (AL VAR) (and obvious rules for integer expressions),

$$\frac{\dfrac{}{y{:}A \vdash y : A{::}T_{\mathsf{res}}(y)} \qquad \dfrac{\vdots}{y{:}A \vdash y.\mathsf{f} + 1 : \mathsf{int}{::}T_{\mathsf{res}}(\mathsf{sel}_{pre}(y,\mathsf{f}) + 1)}}{y{:}A \vdash y.\mathsf{f}{:=}y.\mathsf{f} + 1 : [\,]{::}T_{\mathsf{upd}}(y, \mathsf{f}, \mathsf{sel}_{pre}(y,\mathsf{f}) + 1)}$$

Table 6.3 *Recursive object specifications*

$\|\mathsf{bool}\|_\Sigma$	$\stackrel{def}{=}$	BVal
$\|\top\|_\Sigma$	$\stackrel{def}{=}$	Val
$\|B\|_\Sigma$	$\stackrel{def}{=}$	$\{l \in \mathsf{Loc} \mid \emptyset;\emptyset \vdash \Sigma.l \prec: B\}$ where $B \equiv [\mathsf{f}_i: A_i, \mathsf{m}_j: \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}$
$\|\mu(X)A\|_\Sigma$	$\stackrel{def}{=}$	$\{v \in \mathsf{Val} \mid v \in \|A[\mu(X)A/X]\|_\Sigma\}$

and by (AL VAR) also

$$\frac{}{y{:}A \vdash y : A :: T_{\mathsf{res}}(y)}$$

Using the rule (AL LET) for the sequential composition, the result follows from

$$T_{\mathsf{upd}}(y, \mathsf{f}, \mathsf{sel}_{pre}(y, \mathsf{f})+1)[\mathsf{sel}_{int}/\mathsf{sel}_{post}, \mathsf{alloc}_{int}/\mathsf{alloc}_{post}] \wedge T_{\mathsf{res}}(y)[\mathsf{sel}_{int}/\mathsf{sel}_{pre}, \mathsf{alloc}_{int}/\mathsf{alloc}_{pre}] \to T$$

The idea is thus that proving recursive specifications proceeds by pushing the recursive type sufficiently far "down" and then using the rules of the original system.

6.2 Semantics of Recursive Specifications

6.2.1 Existence of Store Specifications

Next, we adapt our notion of store specification to recursive specifications. The existence proof is very similar to the one given in Section 5.1.

Definition 6.2.1. *A* store specification *is a record* $\Sigma \in \mathsf{Rec}_{\mathsf{Loc}}(RecSpec)$ *such that for all* $l \in dom(\Sigma)$*,*

$$\Sigma.l \quad \equiv \quad \mu(X)[\mathsf{f}_i: A_i, \mathsf{m}_j: \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}$$

is a closed (recursive) object specification. Let RecStSpec be the set of all such store specifications.

Note that because of the (SUBSPEC FOLD) and (SUBSPEC UNFOLD) rules of recursive specifications, the requirement that only object specifications with a μ-binder in head position occur in Σ is no real restriction.

Definition 6.2.2 (Recursive Object Specifications)**.** *Let* $\Sigma \in RecStSpec$*. For closed specifications A define* $\|A\|_\Sigma \subseteq \mathsf{Val}$ *inductively as in Table 6.3. Note that this is a proper definition since, in the last case, the number of binders* μ *in* head position *strictly decreases with each unfolding, due to the formal contractiveness ensured by the syntactic restriction in* (6.3)*.*

The definition is extended to specification contexts $\|\Gamma\|_\Sigma \subseteq \mathsf{Env}$ *as in Definition 5.1.4 on page 79.*

The definition of the functional Φ (cf. Definition 5.1.7) remains virtually the same apart from an unfolding of the recursive specification in the cases for field and method

Table 6.4 *Store predicate for recursive object specifications*

$$
\begin{array}{lll}
\langle \sigma, \phi \rangle \in \Phi(Y,X)_\Sigma & \stackrel{def}{\iff} & (\textsc{Dom})\quad \mathsf{dom}(\Sigma) = \mathsf{dom}(\phi) \wedge \\
& & \qquad \forall l \in \mathsf{dom}(\Sigma).\, \mathsf{dom}(\Sigma.l) = [\mathsf{f}_i{:}A_i,\, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \implies \\
& & \qquad\quad \mathsf{dom}(\phi.l) = \{\mathsf{m}_j\}_{j\in J};\ \text{and} \\
& & \forall l \in \mathsf{dom}(\Sigma)\ \text{where}\ \Sigma.l \equiv [\mathsf{f}_i{:}A_i,\, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}: \\
& & (\mathrm{F})\quad \forall i \in I.\ \ \sigma.l.\mathsf{f}_i \in \|A_i[\Sigma.l/X]\|_\Sigma;\ \text{and} \\
& & (\mathrm{M})\quad \forall j \in J\ \forall \Sigma' \succcurlyeq \Sigma\ \forall \sigma', \sigma'' \in \mathsf{St}\ \forall \phi' \in \mathsf{RSF}\ \forall l' \in \mathsf{Loc}\ \forall v \in \mathsf{Val}. \\
& & \qquad l' \in \|\Sigma.l\|_{\Sigma'}\ \wedge\ \langle \sigma', \phi' \rangle \in Y_{\Sigma'}\ \wedge\ \sigma.l.\mathsf{m}_j(l', \sigma') = \langle v, \sigma'' \rangle \implies \\
& & \qquad\quad \exists \Sigma'' \succcurlyeq \Sigma'\ \exists \phi'' \in \mathsf{RSF}.\ \ \phi.l.\mathsf{m}_j(l', \sigma', \phi', \Sigma') = (\Sigma'', \phi'') \wedge \\
& & (\mathrm{M1})\qquad (\pi_{\mathrm{Val}}(\sigma'), v, \pi_{\mathrm{Val}}(\sigma'')) \in [\![T_j[l'/y_j]]\!] \\
& & (\mathrm{M2})\qquad \langle \sigma'', \phi'' \rangle \in X_{\Sigma''} \\
& & (\mathrm{M3})\qquad v \in ||B_j[\Sigma.l/X][l'/y_j]||_{\Sigma''}
\end{array}
$$

result specifications: In the definition of the domain of choice functions, replace the set of (syntactic) store specifications *StSpec* by *recursive* ones, *RecStSpec*,

$$\mathsf{RSF} \;=\; \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Rec}_{\mathcal{M}}(\mathsf{Loc} \times \mathsf{St} \times \mathsf{RSF} \times \mathit{RecStSpec} \to \mathit{RecStSpec} \times \mathsf{RSF}))$$

Definition 6.2.3. *Let* $\mathcal{R} \stackrel{def}{=} \mathcal{P}(\mathsf{St} \times \mathsf{RSF})^{\mathit{RecStSpec}}$ *denote the collection of families of subsets of* $\mathsf{St} \times \mathsf{RSF}$*, indexed by store specifications (in the sense of Definition 6.2.1). We define a functional* $\Phi : \mathcal{R}^{op} \times \mathcal{R} \to \mathcal{R}$ *in Table 6.4, and write*

$$[\![\Sigma]\!] \stackrel{def}{=} \{\sigma \in \mathsf{St} \mid \exists \phi \in \mathsf{RSF}.\ \langle \sigma, \phi \rangle \in \mathit{fix}(\Phi)_\Sigma\}$$

The proof of Lemma 5.1.8 can be adapted to show that this functional also has a unique fixed point:

Lemma 6.2.4. *Functional* Φ*, defined in Table 6.4, has a unique fixpoint* $\mathit{fix}(\Phi) \in \mathcal{R}$*.*

6.2.2 Semantics

We extend the interpretation of specifications of Section 4.3.4 to the new cases in Table 6.5, where η maps specification variables to admissible subsets of $\mathsf{Val} \times \mathsf{St}$: We write $\eta \vdash \Delta$ if, for all $X \prec: Y$ in Δ, $\eta(X) \subseteq \eta(Y)$.

We briefly observe the following facts, (the duals of) which are standard (Davey and Priestley 2002).

- By Tarski's Fixed Point Theorem, every monotonic map $f : L \to L$ on a complete lattice $(L, \leq)$ has a greatest fixed-point $\mathit{gfp}(f)$ (which is in fact the greatest *post*-fixed point).
- If $f : L \to L$ additionally preserves meets of decreasing countable chains $x_0 \geq x_1 \geq \ldots$, i.e., $f(\bigwedge_i x_i) = \bigwedge_i f(x_i)$, then the greatest fixed point of f can be obtained as

  $$\mathit{gfp}(f) \;=\; \bigwedge\{f^n(\top) \mid n \in \mathbb{N}\}$$

  where $\top$ is the greatest element of L.

Table 6.5 *Interpretation of recursive specifications*

$$
\begin{array}{lcl}
[\![\Gamma;\Delta \vdash X]\!]\,\rho\eta & \stackrel{def}{=} & \eta(X) \\
[\![\Gamma;\Delta \vdash \mathsf{bool}]\!]\,\rho\eta & \stackrel{def}{=} & \mathsf{BVal} \times \mathsf{St} \\
[\![\Gamma;\Delta \vdash \top]\!]\,\rho\eta & \stackrel{def}{=} & \mathsf{Val} \times \mathsf{St} \\
[\![\Gamma;\Delta \vdash \mu(X)A]\!]\,\rho\eta & \stackrel{def}{=} & \mathit{gfp}(\lambda\chi.\,[\![\Gamma;\Delta, X{\prec:}\top \vdash A]\!]\,\rho\eta[X := \chi]) \\
[\![\Gamma;\Delta \vdash B]\!]\,\rho\eta & \stackrel{def}{=} & \left\{ \langle l,\sigma\rangle \;\middle|\; \begin{array}{ll} \text{(i)} & \forall i \in I.\ \langle \sigma.l.\mathsf{f}_i, \sigma\rangle \in [\![\Gamma;\Delta \vdash A_i]\!]\,\rho\eta \\ \text{(ii)} & \forall j \in J.\ \sigma.l.\mathsf{m}_j(l,\sigma) = \langle v,\sigma'\rangle \implies \\ & \qquad \langle v,\sigma'\rangle \in [\![\Gamma, y_j{:}B;\Delta \vdash B_j]\!]\,\rho[y_j := l]\eta \\ & \qquad \wedge\ (\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma, y_j{:}B] \vdash T_j]\!]\,\rho[y_j := l] \end{array} \right\}
\end{array}
$$

where $B \equiv [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$

- For a complete lattice $(L,\leq)$ and any set A, the set of maps $A \to L$ forms a complete lattice when ordered pointwise, with the meet of $\{f_i \mid i \in I\}$ given by $\lambda a.\bigwedge_i f_i(a)$.
- The greatest fixed point operator is monotonic: Suppose $f \leq g$ are monotonic maps in the lattice $L \to L$, then $\mathit{gfp}(f) \leq \mathit{gfp}(g)$.
- Composition preserves meets of descending chains: If $f_0 \geq f_1 \geq \ldots$ and $g_0 \geq g_1 \geq \ldots$ are maps in $L \to L$ s.t. every f_i and g_j is monotonic and preserves meets of descending chains then $\bigwedge_i f_i \circ \bigwedge_j g_j = \bigwedge_n (f_n \circ g_n)$. It follows that $\mathit{gfp}(\bigwedge_i f_i) = \bigwedge_i \mathit{gfp}(f_i)$ i.e., the operator gfp also preserves meets of chains.

Let $\mathcal{A}dm(\mathsf{Val}\times\mathsf{St})$ denote the set of admissible subsets of $\mathsf{Val}\times\mathsf{St}$, i.e., those subsets that are closed under taking least upper bounds of countable chains. Since $\mathcal{A}dm(\mathsf{Val}\times\mathsf{St})$ is closed under arbitrary intersections it forms a complete lattice when ordered by set inclusion. Therefore, the set of specification environments $\eta : \mathit{SpecVar} \to \mathcal{A}dm(\mathsf{Val}\times\mathsf{St})$ endowed with the pointwise ordering forms a complete lattice.

In the following, we show that the interpretation of specifications given above is well-defined. More precisely, we show that meets of descending chains of environments are preserved.

Lemma 6.2.5 (Well-definedness)**.** $[\![\Gamma;\Delta \vdash A]\!]$ *preserves meets of descending chains:*

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

In particular, this result shows that the greatest fixed point used in the definition of recursive specifications in Table 6.5 exists, by the observations on complete lattices and monotonic maps made above.

Proof. We show this lemma by induction on the structure of A. The only interesting case is where A is of the form $\mu(X)B$.

Suppose $\eta_0 \geq \eta_1 \geq \ldots$. If we let $f_i : \mathcal{A}dm(\mathsf{Val}\times\mathsf{St}) \to \mathcal{A}dm(\mathsf{Val}\times\mathsf{St})$,

$$f_i(\chi) \;\stackrel{def}{=}\; [\![\Gamma;\Delta, X \vdash B]\!]\,\rho\eta_i[X := \chi], \qquad i \in \mathbb{N}$$

then the induction hypothesis entails that each f_i is monotonic, and $f_0 \geq f_1 \geq \ldots$ is a descending chain of environments. Moreover, since for each $i \in \mathbb{N}$ and descending chain $\chi_0 \supseteq \chi_1 \supseteq \ldots$ in $\mathcal{A}dm(\mathsf{Val} \times \mathsf{St})$

$$\bigwedge_j \eta_i[X := \chi_j] \quad = \quad \eta_i[X := \bigcap_j \chi_j]$$

the induction hypothesis shows that each f_i preserves meets:

$$\begin{aligned} f_i(\textstyle\bigcap_j \chi_j) &= [\![\Gamma; \Delta, X \vdash B]\!]\, \rho(\textstyle\bigwedge_j \eta_i[X := \chi_j]) \\ &= \textstyle\bigcap_j [\![\Gamma; \Delta, X \vdash B]\!]\, \rho(\eta_i[X := \chi_j]) \\ &= \textstyle\bigcap_j f_i(\chi_j) \end{aligned}$$

We obtain

$$\begin{aligned} [\![\Gamma; \Delta \vdash A]\!]\, \rho(\textstyle\bigwedge_i \eta_i) &= \mathit{gfp}(\lambda\chi.\, [\![\Gamma; \Delta, X \vdash B]\!]\, \rho(\textstyle\bigwedge_i \eta_i)[X := \chi]) && \text{by definition} \\ &= \mathit{gfp}(\lambda\chi.\, [\![\Gamma; \Delta, X \vdash B]\!]\, \rho(\textstyle\bigwedge_i \eta_i[X := \chi])) && \text{pointwise meet} \\ &= \mathit{gfp}(\lambda\chi.\, \textstyle\bigcap_i [\![\Gamma; \Delta, X \vdash B]\!]\, \rho\eta_i[X := \chi]) && \text{by induction} \\ &= \mathit{gfp}(\textstyle\bigwedge_i f_i) && \text{pointwise meet} \\ &= \textstyle\bigcap_i \mathit{gfp}(f_i) && \mathit{gfp} \text{ preserves meet} \\ &= \textstyle\bigcap_i [\![\Gamma; \Delta \vdash A]\!]\, \rho\eta_i && \text{by definition} \end{aligned}$$

which concludes the proof. □

Lemma 6.2.6 (Substitution). *For all* $\Gamma; \Delta, X \vdash A$, $\Gamma; \Delta \vdash B$, ρ *and* η,

$$[\![\Gamma; \Delta, X \vdash A]\!]\, \rho(\eta[X := [\![\Gamma; \Delta \vdash B]\!]\, \rho\eta]) \quad = \quad [\![\Gamma; \Delta \vdash A[B/X]]\!]\, \rho\eta$$

Proof. By induction on A. □

6.3 Soundness

6.3.1 Syntactic Approximations

Recall the statement of Lemma 5.2.1, one of the key lemmas in the proof of the soundness theorem:

$$\forall \sigma, \Sigma, l, A.\ A \text{ closed} \ \wedge\ \sigma \in [\![\Sigma]\!] \ \wedge\ l \in \|A\|_\Sigma \quad \Longrightarrow \quad \langle l, \sigma\rangle \in [\![A]\!] \tag{6.4}$$

In Section 5.2 this 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 Definition 6.2.3 would force a similar unfolding of A in the inductive step, thus not necessarily decreasing the size of A.

Instead, we consider finite approximations as in (Amadio and Cardelli 1993), where we get rid of recursion by unfolding a finite number of times and then replacing all remaining occurrences of recursion by $\top$. We call a specification *non-recursive* if it does not contain any occurrences of specifications of the form $\mu(X)B$.

Table 6.6 *Approximations*

$X\|^{k+1}$	$\stackrel{def}{=}$	X
$\top\|^{k+1}$	$\stackrel{def}{=}$	$\top$
$\mathsf{bool}\|^{k+1}$	$\stackrel{def}{=}$	bool
$\mu(X)A\|^{k+1}$	$\stackrel{def}{=}$	$A[\mu(X)A/X]\|^{k+1}$
$B\|^{k+1}$	$\stackrel{def}{=}$	$[\mathsf{f}_i : A_i\|^k,\ \mathsf{m}_j : \varsigma(y_j)B_j\|^k :: T_j]_{i\in I, j\in J}$

where $B \equiv [\mathsf{f}_i{:}\, A_i,\ \mathsf{m}_j{:}\, \varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$

Definition 6.3.1 (Approximations)**.** *For each $A \in RecSpec$ and $k \in \mathbb{N}$, we define $A|^0 \stackrel{def}{=} \top$ and $A|^{k+1}$ by the cases given in Table 6.6.*

Note that, as in (Amadio and Cardelli 1993), well-definedness of approximation can be shown by a well-founded induction on the lexicographic order on k and the number of μ in head position. In particular recall that our definition of recursive specifications syntactically ruled out troublesome cases like $\mu(X)X$.

Properties of Approximations

Unfortunately, approximations $A|^k$ as defined above do not in fact approximate A (from above) with respect to the subspecification relation $\prec:$, the reason being the invariance in field specifications. For example, if $A \equiv [\mathsf{f}_1 : X, \mathsf{f}_2 : \mathsf{bool}]$ then

$$\begin{aligned}
\mu(X)\mu(Y)A|^2 &= [\mathsf{f}_1 : \mu(X)\mu(Y)A, \mathsf{f}_2 : \mathsf{bool}]|^2 \\
&= [\mathsf{f}_1 : \mu(X)\mu(Y)A|^1, \mathsf{f}_2 : \mathsf{bool}|^1] \\
&= [\mathsf{f}_1 : [\mathsf{f}_1 : \mu(X)\mu(Y)A, \mathsf{f}_2 : \mathsf{bool}]|^1, \mathsf{f}_2 : \mathsf{bool}] \\
&= [\mathsf{f}_1 : [\mathsf{f}_1 : \mu(X)\mu(Y)A|^0, \mathsf{f}_2 : \mathsf{bool}|^0], \mathsf{f}_2 : \mathsf{bool}] \\
&= [\mathsf{f}_1 : [\mathsf{f}_1 : \top, \mathsf{f}_2 : \top], \mathsf{f}_2 : \mathsf{bool}]
\end{aligned}$$

By inspection of the rules, $\vdash \mu(X)\mu(Y)A \prec: \mu(X)\mu(Y)A|^2$ requires to show

$$\Gamma;\Delta \vdash [\mathsf{f}_1 : [\mathsf{f}_1 : \mu(X)\mu(Y)A, \mathsf{f}_2 : \mathsf{bool}], \mathsf{f}_2 : \mathsf{bool}] \prec: [\mathsf{f}_1 : [\mathsf{f}_1 : \top, \mathsf{f}_2 : \top], \mathsf{f}_2 : \mathsf{bool}] \qquad (6.5)$$

for appropriate Γ and Δ. But subspecifications of object specifications can only be derived for equal components f_1, by the rule (SUBSPEC OBJ) of Section 4.3.In particular, $\mathsf{bool} \prec: \top$ does not imply that (6.5) holds.

For this reason we consider the more generous subspecification relation that also allows for subspecifications in field components, by replacing the rule (SUBSPEC OBJ) for object specifications with (SUBSPEC OBJ'), given in Table 6.7.

We write $\stackrel{\mathcal{F}}{\prec}:$ for this relation, and observe that

$$\vdash A \prec: B \quad \Longrightarrow \quad \vdash A \stackrel{\mathcal{F}}{\prec}: B \qquad (6.6)$$

Table 6.7 *The generalised object subspecification rule*

$$\text{(SUBSPEC OBJ')}\quad \frac{\begin{array}{llll} \Gamma\quad;\Delta \vdash A_i \prec: A'_i & \forall i \in I' & \Gamma\quad;\Delta \vdash A_i & \forall i \in I \\ \Gamma, y_j{:}\top;\Delta \vdash B_j \prec: B'_j & \forall j \in J' & \Gamma, y_j{:}\top;\Delta \vdash B_j & \forall j \in J \\ \vdash_{\mathsf{fo}} T_j \to T'_j & \forall j \in J' & [\Gamma, y_j{:}\top] \vdash T_j & \forall j \in J \qquad I' \subseteq I \\ & & [\Gamma, y_j{:}\top] \vdash T'_j & \forall j \in J' \qquad J' \subseteq J \end{array}}{\Gamma;\Delta \vdash [\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} \prec: [\mathsf{f}_i{:}A'_i,\ \mathsf{m}_j{:}\varsigma(y_j)B'_j{::}T'_j]_{i\in I', j\in J'}}$$

The relation $\overset{\mathcal{F}}{\prec}:$ is still sufficient to guarantee soundness with respect to the semantics $[\![A]\!]$ of Section 6.2.2, as shown in Lemma 6.3.4 below. First, we obtain the following approximation lemma for the $\overset{\mathcal{F}}{\prec}:$ relation, justifying the name of the operation $A|^k$.

Lemma 6.3.2 (Approximation)**.** *For all specifications $\Gamma;\Delta \vdash A$, the following hold.*

1. *For all $k \in \mathbb{N}$, $\Gamma;\Delta \vdash A \overset{\mathcal{F}}{\prec}: A|^k$.*
2. *For all $k, l \in \mathbb{N}$, $\Gamma;\Delta \vdash A|^{k+l} \overset{\mathcal{F}}{\prec}: A|^k$.*
3. *If A is non-recursive then there exists $n \in \mathbb{N}$ such that for all $k \geq n$, $A \equiv A|^k$.*

Sketch of proof. The proofs are by induction on the lexicographic order on k and the number of binders μ in head position in A, then considering cases for the specification A. □

Lemma 6.3.3 (Monotonicity of $(\cdot)|^k$)**.** *For all Γ, Δ, A, B and $k \in \mathbb{N}$,*

$$\Gamma;\Delta \vdash A \overset{\mathcal{F}}{\prec}: B \quad\Longrightarrow\quad \Gamma;\Delta \vdash A|^k \overset{\mathcal{F}}{\prec}: B|^k$$

Proof. By induction on the derivation of $\Gamma;\Delta \vdash A \overset{\mathcal{F}}{\prec}: B$. The case for (SUBSPEC TOP) is immediate. The cases for (SUBSPEC FOLD) and (SUBSPEC UNFOLD) follow by reflexivity of $\overset{\mathcal{F}}{\prec}:$, since $\mu(X)A|^k \equiv A[(\mu(X)A)/X]|^k$ for all k. The case for (SUBSPEC OBJ') follows by applying the induction hypothesis to all component specifications.

Finally, in the case (SUBSPEC VAR), necessarily $A \equiv X$ and there exists an assumption $X \prec: B$ in Δ. Without loss of generality let $k > 0$. By definition, $X|^k \equiv X$. By Lemma 6.3.2 part (1), $\Gamma;\Delta \vdash B \overset{\mathcal{F}}{\prec}: B|^k$. Thus by transitivity also $\Gamma;\Delta \vdash X|^k \overset{\mathcal{F}}{\prec}: B|^k$. □

Soundness of Subspecification

Soundness of subspecification is established next:

Lemma 6.3.4 (Soundness of $\overset{\mathcal{F}}{\prec}:$)**.** *If $\Gamma;\Delta \vdash A \overset{\mathcal{F}}{\prec}: B$, $\rho \in \mathsf{Env}$ and $\eta \vDash \Delta$ then*

$$[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \subseteq [\![\Gamma;\Delta \vdash B]\!]\,\rho\eta$$

Proof. By induction on the derivation of $\Gamma;\Delta \vdash A \overset{\mathcal{F}}{\prec}: B$. We consider cases depending on the last rule applied in this derivation.

- The cases for reflexivity and transitivity are immediate.

- Case (TOP).
  By inspection of the rule, necessarily $B \equiv \top$, and from the semantics

$$[\![\Gamma; \Delta \vdash \top]\!]\,\rho\eta \;=\; \mathsf{Val} \times \mathsf{St} \supseteq [\![\Gamma; \Delta \vdash A]\!]\,\rho\eta$$

  for all A.

- Cases (FOLD) and (UNFOLD).
  These cases follow from the fact that the denotation of $\mu(X)A$ is indeed a fixed point,

$$\begin{array}{lcll} [\![\Gamma; \Delta \vdash \mu(X)A]\!]\,\rho\eta & = & \mathit{gfp}(\lambda\chi.\,[\![\Gamma; \Delta, X \vdash A]\!]\,\rho\eta[X = \chi]) & \text{by definition} \\ & = & [\![\Gamma; \Delta, X \vdash A]\!]\,\rho(\eta[X = [\![\Gamma; \Delta \vdash \mu(X)A]\!]\,\rho\eta]) & \text{fixed point} \\ & = & [\![\Gamma; \Delta \vdash A[\mu(X)A/X]]\!]\,\rho\eta & \text{Subst. Lemma 6.2.6} \end{array}$$

- Case (SUBSPEC VAR).
  From the conclusion of the rule, there is some specification variable X such that $A \equiv X$ and $X \prec: B$ occurs in Δ. Recall that the only upper bounds in Δ are of the form Y or $\top$, for $Y \in \mathit{SpecVar}$. In the latter case the conclusion is trivial, in the former, it follows since

$$\eta(X) \;\subseteq\; \eta(Y)$$

  holds by assumption $\eta \vDash \Delta$.

- Case (SUBSPEC OBJ').
  By inspection of the rule (SUBSPEC OBJ') we must have

$$A \quad\equiv\quad [\mathsf{f}_i : A_i,\, \mathsf{m}_j : \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}$$

  and

$$B \quad\equiv\quad [\mathsf{f}_i : A'_i,\, \mathsf{m}_j : \varsigma(y_j)B'_j :: T'_j]_{i\in I', j\in J'}$$

  such that for all $i \in I'$ and $j \in J'$, $\Gamma; \Delta \vdash A_i \overset{\mathcal{F}}{\prec:} A'_i$ and $\Gamma, y_j{:}\top; \Delta \vdash B_j \overset{\mathcal{F}}{\prec:} B'_j$ and $\vdash_{\mathsf{fo}} T_j \to T'_j$. By induction hypothesis,

$$[\![\Gamma; \Delta \vdash A_i]\!]\,\rho\eta \subseteq [\![\Gamma; \Delta \vdash A'_i]\!]\,\rho\eta$$

  and

$$[\![\Gamma, y_j; \Delta \vdash B_j]\!]\,(\rho[y_j := l])\eta \subseteq \left[\!\!\left[\Gamma, y_j; \Delta \vdash B'_j\right]\!\!\right](\rho[y_j := l])\eta$$

  for all $i \in I'$, $j \in J'$ and $l \in \mathsf{Loc}$. Moreover, by soundness of $\vdash_{\mathsf{fo}}$ we know

$$[\![[\Gamma], y_j \vdash T_j]\!]\,(\rho[y_j := l]) \subseteq \left[\!\!\left[[\Gamma], y_j \vdash T'_j\right]\!\!\right](\rho[y_j := l])$$

  for $j \in J'$. So by definition of $[\![\Gamma; \Delta \vdash A]\!]$ and $[\![\Gamma; \Delta \vdash B]\!]$,

$$\langle l, \sigma\rangle \in [\![\Gamma; \Delta \vdash A]\!]\,\rho\eta \quad\Longrightarrow\quad \langle l, \sigma\rangle \in [\![\Gamma; \Delta \vdash B]\!]\,\rho\eta$$

- Case (SUBSPEC REC).
  Suppose that $\Gamma;\Delta \vdash \mu(X)A \overset{\mathcal{F}}{\prec:} \mu(Y)B$ has been derived from the premiss

  $$\Gamma;\Delta, Y \prec: \top, X \prec: Y \vdash A \overset{\mathcal{F}}{\prec:} B$$

  We use the fact that $[\![\Gamma;\Delta \vdash \mu(Y)B]\!]\,\rho\eta$ is the greatest post-fixed point of the map f where

  $$f(\chi) \overset{def}{=} [\![\Gamma;\Delta, Y \vdash B]\!]\,\rho\eta[Y := \chi]$$

  which is monotonic according to Lemma 6.2.5. Since $\alpha \overset{def}{=} [\![\Gamma;\Delta \vdash \mu(X)A]\!]\,\rho\eta$ is a fixed point of $\lambda\chi.\,[\![\Gamma;\Delta, X \vdash A]\!]\,\rho\eta[X := \chi]$ we calculate

  $$\begin{array}{rcll} \alpha & = & [\![\Gamma;\Delta, Y, X \prec: Y \vdash A]\!]\,\rho\eta' & \Gamma;\Delta, X \vdash A \text{ independent of } \eta(Y) \\ & \subseteq & [\![\Gamma;\Delta, Y, X \prec: Y \vdash B]\!]\,\rho\eta' & \text{by induction} \\ & = & f(\alpha) & \Gamma;\Delta, Y \vdash B \text{ independent of } \eta(X) \end{array}$$

  where $\eta' \overset{def}{=} \eta[X := \alpha, Y := \alpha]$, which shows α is a post-fixed point of f. Note that we can apply induction in the second line since $\eta'(X) \subseteq \eta'(Y)$ holds. Hence,

  $$[\![\Gamma;\Delta \vdash \mu(X)A]\!]\,\rho\eta \;=\; \alpha \;\subseteq\; gfp(f) \;=\; [\![\Gamma;\Delta \vdash \mu(Y)B]\!]\,\rho\eta$$

  as required.

□

Remark 6.3.5. *While we have just established soundness of* $\overset{\mathcal{F}}{\prec:}$ *with respect to the semantics* $[\![A]\!]$ *of specifications as defined in Table 6.5, it should be noted that this is not the case for the semantics* $\|A\|$ *(cf. Table 6.3). In particular, the relation* $\overset{\mathcal{F}}{\prec:}$ *is* not *sufficient to establish an invariance result corresponding to the key Lemma 5.2.5; the obvious problems are the combination of field update and weakening along field specifications permitted through rule* (SUBSPEC OBJ')*.*

A crucial difference between $[\![A]\!]$ *and* $\|A\|$ *is that* $[\![A]\!]$ *is defined purely in terms of field selection and method invocation. This provides an explanation why Lemma 6.3.4 could be established.* □

Relating Semantics and Syntactic Approximations

Together, Lemma 6.3.4 and Lemma 6.3.2(1) show $[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \subseteq [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$ for all $\eta \vDash \Delta$ and $k \in \mathbb{N}$; in particular,

$$[\![A]\!]\,\eta \;\subseteq\; \bigcap_{k\in\mathbb{N}} [\![A|^k]\!]\,\eta \tag{6.7}$$

for closed specifications A. For the reverse inclusion, we use the characterisation of greatest fixed points as meet of a descending chain, which is in close correspondence with the syntactic approximations.

Lemma 6.3.6 (Combining Substitution and Approximation)**.** *For all specifications A, B, all X such that* $\Gamma;\Delta \vdash B$ *and* $\Gamma;\Delta, X \vdash A$*, and for all* $k, l \in \mathbb{N}$

$$\Gamma;\Delta \vdash A[B/X]|^l \overset{\mathcal{F}}{\prec:} A[B|^k/X]|^l$$

In particular, $[\![\Gamma;\Delta \vdash A[B/X]|^l]\!]\,\rho\eta \subseteq [\![\Gamma;\Delta \vdash A[B|^k/X]|^l]\!]\,\rho\eta$ *for all* $\eta \vDash \Delta$*, by Lemma 6.3.4.*

Proof. The proof is by induction on the lexicographic order on l and the number of μ in head position.

- Case $l = 0$. By definition, $A[B/X]|^0 \equiv \top$ and $A[B|^k/X]|^0 \equiv \top$. Thus

$$\Gamma;\Delta \vdash A[B/X]|^0 \overset{\mathcal{F}}{\prec:} A[B|^k/X]|^0$$

  follows immediately from reflexivity of the $\overset{\mathcal{F}}{\prec:}$ relation.

- Case $l > 0$. We consider possible cases for A:

  - A is X. By definition of substitution, $A[B/X]|^l \equiv B|^l$ and $A[B|^k/X]|^l \equiv B|^k|^l$. By Lemma 6.3.2 part (1), $\Gamma;\Delta \vdash B \overset{\mathcal{F}}{\prec:} B|^k$. Thus

$$\Gamma;\Delta \vdash B|^l \overset{\mathcal{F}}{\prec:} B|^k|^l$$

    follows by Lemma 6.3.3.

  - A is $\top$, bool or $Y \neq X$. Then $\vdash A[B/X]|^l \equiv A|^l$ and $A[B|^k/X]|^l \equiv A|^l$. Thus

$$\Gamma;\Delta \vdash A[B/X]|^l \overset{\mathcal{F}}{\prec:} A[B|^k/X]|^l$$

    follows by reflexivity of the $\overset{\mathcal{F}}{\prec:}$ relation.

  - A is $[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$. Then, by induction hypothesis,

$$\Gamma;\Delta \vdash A_i[B/X]|^{l-1} \overset{\mathcal{F}}{\prec:} A_i[B|^k/X]|^{l-1}$$

    and

$$\Gamma, y_j{:}A;\Delta \vdash B_j[B/X]|^{l-1} \overset{\mathcal{F}}{\prec:} B_j[B|^k/X]|^{l-1}$$

    for all $i \in I$ and $j \in J$. Hence, by definition of $\overset{\mathcal{F}}{\prec:}$ and $(\cdot)|^l$

$$\Gamma;\Delta \vdash A[B/X]|^l \overset{\mathcal{F}}{\prec:} [\mathsf{f}_i : A_i[B|^k/X], \mathsf{m}_j : B_j[B|^k/X]]|^l$$

    By definition, $[\mathsf{f}_i : A_i[B|^k/X], \mathsf{m}_j : B_j[B|^k/X]]|^l \equiv A[B|^k/X]|^l$ from which the statement of the Lemma follows.

  - A is $\mu(Y)C$, without loss of generality Y not free in B. Then by induction hypothesis we find $\Gamma;\Delta \vdash C[A/Y][B/X]|^l \overset{\mathcal{F}}{\prec:} C[A/Y][B|^k/X]|^l$. Using properties of syntactic substitutions, we calculate

$$\begin{array}{rcll} A[B/X]|^l & \equiv & \mu(Y)(C[B/X])|^l & \text{substitution} \\ & \equiv & C[B/X][(\mu(Y)(C[B/X]))/Y]|^l & \text{definition of } (\cdot)|^l \\ & \equiv & C[B/X][(A[B/X])/Y]|^l & \text{substitution} \\ & \equiv & C[A/Y][B/X]|^l & Y \notin \mathsf{fv}(B) \end{array}$$

    and analogously $C[A/Y][B|^k/X]|^l \equiv A[B|^k/X]|^l$, which entails the result.

□

Lemma 6.3.7 (Approximation of Specifications)**.** *For all* $\Gamma;\Delta \vdash A$*, for all* $\rho \in \mathsf{Env}$ *and environments* $\eta \vDash \Delta$*,*

$$[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta = \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$$

Proof. By (6.7), all that remains to show is $[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \supseteq \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$. We proceed by induction on the lexicographic order on pairs (M, A) where M is an upper bound on the number of μ-binders in A. For the base case, $M = 0$, by Lemma 6.3.2(3) there exists $n \in \mathbb{N}$ such that for all $k \geq n$, $A|^k \equiv A$, and so in fact

$$[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \;=\; [\![\Gamma;\Delta \vdash A|^n]\!]\,\rho\eta \;\supseteq\; \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$$

Now suppose that A contains at most $M+1$ μ binders. We consider cases for A.

- Case A is $\top, X$ or bool. Then as above, there exists $n \in \mathbb{N}$ such that for all $k \geq n$, $A|^k \equiv A$ and we are done.
- Case A is $[\mathsf{f}_i\colon A_i,\ \mathsf{m}_j\colon \varsigma(y_j)B_j :: T_j]_{i\in I, j\in J}$. Then, by induction hypothesis,

  $$[\![\Gamma;\Delta \vdash A_i]\!]\,\rho\eta \;\supseteq\; \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A_i|^k]\!]\,\rho\eta$$

  and

  $$[\![\Gamma, y_j;\Delta \vdash B_j]\!]\,(\rho[y_j := l])\eta \;\supseteq\; \bigcap_{k\in\mathbb{N}} [\![\Gamma, y_j;\Delta \vdash B_j|^k]\!]\,(\rho[y_j := l])\eta$$

  for $i \in I$ and $j \in J$. Hence, if $\langle l, \sigma\rangle \in [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$ for all $k \in \mathbb{N}$ then

  $$\langle \sigma.l.\mathsf{f}_i, \sigma\rangle \in \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A_i|^k]\!]\,\rho\eta \;\subseteq\; [\![\Gamma;\Delta \vdash A_i]\!]\,\rho\eta$$

  and $\sigma.l.\mathsf{m}_j(l,\sigma) = \langle \nu, \sigma'\rangle$ implies

  $$\langle \nu, \sigma'\rangle \in \bigcap_{k\in\mathbb{N}} [\![\Gamma, y_j;\Delta \vdash B_j|^k]\!]\,(\rho[y_j := l])\eta \subseteq [\![\Gamma, y_j;\Delta \vdash B_j]\!]\,(\rho[y_j := l])\eta$$

  by the definition of $[\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$. This shows $\langle l, \sigma\rangle \in [\![\Gamma;\Delta \vdash A]\!]\,\rho\eta$ as required.
- Case A is $\mu(X)B$. Recall that

  $$[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \;=\; \mathit{gfp}(f_A)$$

  is the greatest *post*-fixed point of $f_A(\chi) \stackrel{def}{=} [\![\Gamma;\Delta, X \vdash B]\!]\,\rho\eta[X := \chi]$. We show that $\alpha \stackrel{def}{=} \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$ is a post-fixed point of f_A, from which

  $$[\![\Gamma;\Delta \vdash A]\!]\,\rho\eta \;\supseteq\; \bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$$

  then follows: First note that by Lemma 6.3.2(2) and Lemma 6.3.4,

  $$\eta[X := [\![\Gamma;\Delta \vdash A|^0]\!]\,\rho\eta] \;\geq\; \eta[X := [\![\Gamma;\Delta \vdash A|^1]\!]\,\rho\eta] \;\geq\; \ldots$$

  forms a countable descending chain of environments. Hence we can calculate

$f_A(\alpha)$	$=$	$[\![\Gamma;\Delta, X \vdash B]\!]\,\rho\eta[X := \alpha]$	def. of f_A
	$=$	$[\![\Gamma;\Delta, X \vdash B]\!]\,\rho(\bigwedge_k \eta[X := [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta])$	def. of α and meets
	$=$	$\bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta, X \vdash B]\!]\,\rho\eta[X := [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta]$	Lemma 6.2.5, meets
	$=$	$\bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash B[A|^k/X]]\!]\,\rho\eta$	substitution, Lemma 6.2.6
	$\supseteq$	$\bigcap_{k\in\mathbb{N}}\bigcap_{l\in\mathbb{N}} [\![\Gamma;\Delta \vdash B[A|^k/X]|^l]\!]\,\rho\eta$	induction hypothesis
	$\supseteq$	$\bigcap_{m\in\mathbb{N}} [\![\Gamma;\Delta \vdash B[A/X]|^m]\!]\,\rho\eta$	Lemma 6.3.6, Lemma 6.3.4
	$=$	$\bigcap_{k\in\mathbb{N}} [\![\Gamma;\Delta \vdash A|^k]\!]\,\rho\eta$	def. of $\mu(X)A|^k$

  i.e., $\alpha \subseteq f_A(\alpha)$. Note that we can apply induction in the fourth line since $A|^k$ does not contain any μ and therefore $B[A|^k/X]$ contains at most M μ-binders.

This concludes the proof. □

6.3.2 Soundness Theorem

After the technical development in the preceding subsection we can now prove (6.4). From this result the soundness proof of the logic extended with recursive specifications then follows, along the lines of the proof presented in Section 5.2 for non-recursive specifications.

Lemma 6.3.8. *For all* $\sigma \in \mathsf{St}$, $\Sigma \in \mathit{RecStSpec}$, $l \in \mathsf{Loc}$ *and closed* $A \in \mathit{RecSpec}$,

$$\sigma \in [\![\Sigma]\!] \ \wedge \ l \in \|A\|_\Sigma \quad \Longrightarrow \quad \langle l, \sigma\rangle \in [\![A]\!]$$

Proof. The proof proceeds by considering finite specifications first. This can be proved by induction on A, as in Lemma 5.2.1. When applying the induction hypothesis we use the fact that $\vdash A \prec: B$ implies $\vdash A \overset{\mathcal{F}}{\prec}: B$, and apply Lemma 6.3.4.

In order to extend the proof to all (possibly recursive) specifications, note that by Lemma 6.3.2 part (1), $\vdash A \overset{\mathcal{F}}{\prec}: A|^k$ holds for all $k \in \mathbb{N}$. By assumption $l \in \|A\|_\Sigma$ this entails $\vdash \Sigma.l \overset{\mathcal{F}}{\prec}: A|^k$ for all k by transitivity. Every $A|^k$ is non-recursive, so by the above considerations, $\langle l, \sigma\rangle \in [\![A|^k]\!]$ for all k. Thus

$$\langle l, \sigma\rangle \in \textstyle\bigcap_{k\in\mathbb{N}} [\![A|^k]\!] \ = \ [\![A]\!]$$

by Lemma 6.3.7. □

For specifications without free specification variables we obtain

Theorem 6.3.9 (Soundness)**.** *If* $\Gamma; \emptyset \vdash a : A :: T$ *then* $\Gamma; \emptyset \vDash a : A :: T$.

Proof Sketch. Firstly, the key lemma 5.2.5 has to be proved with respect to recursive specifications (without free specification variables) and the corresponding store specifications. Apart from the occasional folding and unfolding of recursive specifications this proceeds as before.

Once this result is established, the proof is analogous to Theorem 5.2.6: Let $\Sigma \in \mathit{RecStSpec}$, let $\rho \in \|\Gamma\|_\Sigma$ and suppose $\sigma \in [\![\Sigma]\!]$, i.e., $\langle \sigma, \phi\rangle \in \mathit{fix}(\Phi)_\Sigma$ for some ϕ. Now suppose that a terminates, i.e., there are $v \in \mathsf{Val}$ and $\sigma' \in \mathsf{St}$ such that $[\![a]\!]\rho\sigma = \langle v, \sigma'\rangle$. Thus, by the above, there exists $\phi_a \in \mathsf{SF}$ such that for all $l \in \mathsf{Loc}$, $\phi_a(l, \sigma, \phi, \Sigma) = \langle \Sigma', \phi'\rangle$ for some Σ' and ϕ' where $\Sigma' \succcurlyeq \Sigma$ and $\langle \sigma', \phi'\rangle \in \mathit{fix}(\Phi)_{\Sigma'}$. In particular, $\sigma' \in [\![\Sigma']\!]$. Furthermore,

- $v \in \|A[\rho/\Gamma]\|_{\Sigma'}$
- $(\pi_{\mathrm{Val}}(\sigma), v, \pi_{\mathrm{Val}}(\sigma')) \in [\![[\Gamma] \vdash T]\!]\rho$

In the cases where A is bool or $\top$ one easily obtains $\langle v, \sigma'\rangle \in [\![[\Gamma] \vdash A]\!]\rho$ from the definition of $\|A[\rho/\Gamma]\|_{\Sigma'}$. Otherwise (since by assumption it cannot be a specification variable) A must be a (recursive) object specification. But then Lemma 6.3.8 already entails $\langle v, \sigma'\rangle \in [\![A[\rho/\Gamma]]\!] = [\![[\Gamma] \vdash A]\!]\rho$. □

Chapter 7

Discussion

The study of Abadi-Leino logic from a denotational viewpoint has not been carried for out the belief that this particular logic is the best one can devise. However, it was the first (and, to the best of our knowledge, so far the only) logic for the object-calculus and thus seemed an ideal starting point for this line of research.

This short chapter summarises our technical results about Abadi and Leino's logic. Remaining open questions and avenues for future work are discussed.

7.1 Comparison to Previous Work

7.1.1 A Comparison to Abadi and Leino's Proof

One clear advantage of the soundness theorem (Theorem 5.2.6) over the original soundness proof presented in (Abadi and Leino 2004) is that its content is immediately evident. We claim that this is not the case for Abadi and Leino's soundness theorem, which was stated with respect to an operational semantics of the object calculus. It takes the form of a subject reduction theorem, in the sense that if a judgement is derivable for a term, then it is also derivable for all of its reducts.

Such a syntactic approach works well for type soundness and for Hoare logics of simpler languages, where the meaning of assertions is easy to define. For an example see the exposition in (Winskel 1993, Chapter 6.5) for a while-language where integers are the only storable values. However, we believe the applicability of this method in the presence of higher-order store is questionable: Since (run time) values may provide links to objects in the store, the statement of the subject reduction theorem must actually incorporate locations and thus code in the store. Consequently it must refer to the external concept of store specifications. In contrast to our work, Abadi and Leino did not define a semantics for store specifications independent of derivability by the proof rules (note that our definition of $[\![\Sigma]\!]$ relies on subspecification derivations, but *not* on the judgement derivation relation). This is not very satisfactory, as soundness of the proof system had not been established at that point.

In essence, Abadi and Leino's theorem only establishes the property:

> If $\Gamma \vdash [\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)b_j]_{i\in I, j\in J} : [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J} :: \ldots$ then executing this program yields a location as result (if it terminates) and changes the store such that $[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j]_{i\in I, j\in J}$ can be derived for the object stored at the result location.

This does not "explain" much, in that object specifications (except those corresponding to base type) have no "meaning" independent of the operational semantics: As usual with a subject reduction theorem, one obtains soundness only in the sense that verified programs do not lead to an error situation.

We conjecture that the reason why Abadi and Leino's proof nevertheless has some semantic content is that, at least in the absence of recursive specifications, derivability of a complex specification is based on derivability of its constituent components, so that eventually meaning can be given in terms of basic specifications such as bool. Then, for closed programs of base type, one can define a syntactic characterisation of validity. However, if this is indeed the case, it remains implicit in the soundness theorem of (Abadi and Leino 2004).

7.1.2 Store Specifications as Possible Worlds

While our original intention was to start from the untyped denotational semantics of Chapter 3 and then interpret the specifications A of Abadi and Leino's logic as predicates, exactly as in (Reus and Streicher 2004), we found it necessary to introduce the separate concept of store specifications from (Abadi and Leino 2004) also in our semantic approach. The reason lies in the design of the logic that implicitly assumes, and guarantees, that fields of objects can only be updated with values preserving the original specification. Since objects are kept in the heap we needed a way to keep track of their respective specifications.

Due to their syntactic nature (cf. the discussion at the beginning of Section 5.1), store specifications Σ look like an ad-hoc notion introduced for the sole purpose of obtaining soundness. We later realised that they relate to an established concept: As mentioned in Remark 5.1.5 on page 79, store specifications can be seen as Kripke style possible worlds, indexing sets of untyped values $\|A\|_\Sigma$. A transition to a larger world $\Sigma' \succcurlyeq \Sigma$ extends (possibly strictly) this set of values, $\|A\|_{\Sigma'} \supseteq \|A\|_\Sigma$.

Much of our work relies on the fact that store specifications allow us to characterise these sets of values $\|A\|_\Sigma$ independent of concrete stores. The import of Lemma 5.2.1 on page 87 is to relate such a set $\|A\|_\Sigma$ to the predicate $[\![A]\!]$ we are really interested in, and which *is* defined over $\mathsf{Val} \times \mathsf{St}$. In fact, much of the work in Chapter 6 is concerned with establishing a similar lemma for recursive specifications.

With hindsight, finding a possible worlds semantic structure in our model is not really surprising, as many other semantic models of languages with dynamic allocation of heap store are possible worlds models, too (Levy 2004; Reddy and Yang 2004; Stark 1998). In particular, Levy (2004) uses store worlds that are very similar to our store specifications, but built over *simple types* rather than specifications. He obtains a typed semantics of a higher-order language with general storage, as in the programming language ML. In Part III

we extend his model with subtyping, and arrive at a typed model of imperative objects. Interestingly, in order to prove a coherence result, we will again employ a possible worlds model over *untyped* terms, with predicates comparable to $\|A\|_\Sigma$ here.

Stark's (1998) worlds contain only the allocated locations without any additional information, since he is modelling the ν-calculus that only provides references to unit values. Reddy and Yang (2004) use much more complex *relational correspondences* consisting of *renamings* between visible locations and a relation (itself parameterised over extensions of renamings) to express representation invariants between hidden locations. The order on relational correspondences not only reflects the allocation of new memory but also the "leaking" of previously hidden private locations. These possible worlds are formalised as *subsumptive reflexive graphs* (O'Hearn and Tennent 1995).

Finally, we note that the possible worlds approach, at least with regards to higher-order store as in our and Levy's models, exploits the fact that worlds are closed types and specifications, respectively. This can be seen in our Definition 5.1.4 (on page 79) in the case for object specifications; in Levy's model, a corresponding case uses syntactic equality '$\equiv$' in place of subspecification '$\prec:$'. In the case of store specifications we heavily depended on the fact that syntactic descriptions of all semantic values (i.e., locations and booleans) are available, in order to substitute for free variables in specifications. This will no longer work for the model in Chapter 8 where values also contain higher-order functions. We consider a proper investigation of type dependency and specifications for such a language with higher-order values an important, and challenging, topic for future research.

7.2 Outlook: Towards More Expressive Logics

One long-term objective is to design a better, more powerful and more complete logic building on the lessons learnt from analysing Abadi-Leino logic. To that end, we think one should investigate the following extensions and changes to Abadi and Leino's calculus (above we have already considered one such extension, in the form of recursive specifications).

Local Store. In this thesis we have worked with a global store model. In the semantics, every object, including its fields and methods, is visible to any other object. For Abadi-Leino logic this was sufficient but one significant feature of object-oriented programs is *encapsulation*. Clearly encapsulation is modelled only by a refined notion of store – and accordingly more refined store specifications. Reddy and Yang (2004) and Benton and Leperchey (2005) have presented such models for higher-order languages with storable references but no higher-order store. These models give rise to a large number of correct program equivalences. The authors expressed the need to extend their models to a full object-oriented language and to specifications. Coming from the other end, we have a logic for a simple object-oriented language, but need to incorporate locality and encapsulation.

A complementary approach to finding better (and, most likely, more intricate) models is to restrict the language by imposing a *confinement* condition on programs. Banerjee

and Naumann (2002) use this approach to prove representation independence for a class-based Java-like language. It should be interesting to try similar techniques for a language with higher-order store, such as the object calculus, in order to prove a restricted form of encapsulation and representation independence.

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 by object specifications. In particular, specifying fields in object specifications is limited. For instance recall the bank account from Example 3.3.2. Invariants like $\mathsf{balance} \geq 0$, stating that an account comes without overdraft, cannot be formulated. Note that such an axiom in a transition specification only guarantees that the *current* balance is positive. Also note that it would be possible to extend the logic (and soundness proof) with a subspecification $\mathsf{pos} \prec: \mathsf{int}$, but demanding $\mathsf{balance} : \mathsf{pos}$ may be too strong a requirement: In general it could be necessary to temporarily invalidate the invariant within local method bodies.

Using a store with local fields as described above, it should become possible to accommodate such invariants. Invariance of such a field has to be established only for those methods that can see it.

Method Parameters. Formal method parameters of the form $x{:}A$ can be attached to method specifications – for instance, in the bank account example it would be natural to have a method $\mathtt{deposit}{:}\varsigma(y)\forall(x{:}A)[]{::}T_{\mathsf{deposit}}(y,x)$ – 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. Note that this means that the range of possible inputs varies with the store specification. There are limitations, however, as the resulting object specification B containing such a method specification must still allow for subspecifications. Thus, its semantics $[\![B]\!]$ should not be defined by a recursion with negative occurrences of store.

Dynamic Loading. Dynamic loading of objects is, in a way, already available in the object calculus (and this is one of its advantages over class-based languages). Loading an object of which one only knows its specification $A = [\mathsf{f}_i : A_i; \mathsf{m}_j : \varsigma(x_j)B_j :: T_j]$ corresponds to using a command of which one only knows its result specification A. Thus,

$$x : [\mathsf{load} : \varsigma(y)A :: \exists\overline{z}.\ T_{\mathsf{obj}}(\overline{\mathsf{f}} = \overline{z})] \vdash x.\mathsf{load} : A :: \exists\overline{z}.\ T_{\mathsf{obj}}(\overline{\mathsf{f}} = \overline{z})$$

describes dynamic loading where x might be thought of as class loader and the load command is $x.\mathsf{load}$. It can be used to load any object fulfilling specification A, provided to us "by the context".

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. As outlined at the beginning of Section 6 they are needed to implement recursive datatypes such as lists and trees. In the preceding section we have discussed in full detail how the logic can be safely enriched by recursive specifications.

Parametric Method Specifications. Transition specifications in Abadi-Leino logic cannot refer to methods. This is unnecessary as methods (and their specifications) are fixed at object introduction and assumed to be immutable afterwards. But for programs that, for example, use delegations similar to the *Command* pattern described in (Gamma, Helm, Johnson, and Vlissides 1995), this is not quite adequate: The specification in use is not known at the time of object creation but only at update (and it may change with further updates). As a remedy one could allow placeholders for specifications (B and T) that can be shared inside objects. For example let X and Y be such placeholders then

$$[\mathsf{f} : [\mathsf{n} : \varsigma(x)X::Y], \mathsf{m} : \varsigma(x)X::Y]$$

states that m satisfies whatever specification n satisfies. Note that only n can be updated via f. The invariance of specification still holds, it is just that every object providing a method n will meet specification $[\mathsf{n} : \varsigma(x)X::Y]$ and m will still satisfy $\mathsf{m} : \varsigma(x)X::Y$ if it is implemented as $\mathsf{m} = \varsigma(y)x.\mathsf{f}.\mathsf{n}$. More general transition specifications for m are conceivable that assume Y to hold only for certain calls of n. To find the right (syntactic) restrictions for these specifications, the restrictions revealed by the existence theorem may be helpful.

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 (Gamma, Helm, Johnson, and Vlissides 1995). 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 in the logic and cannot be used later. More useful would be a "behavioural" update where result and transition specifications of the overriding method can be proper subspecifications of the original method. Since the design of Abadi-Leino logic relies on the idea of invariance, any relaxation of this is problematic.

Invariance of Store Specifications. The previous point shows the need for a logic where not all object specifications are preserved. The conclusion we draw from this fact is that it will be worthwhile to develop a calculus where invariance properties are made explicit in the logic. Even though this may clutter proofs (for *users* of such a logic), it may reveal limitations of logics with higher-order *dynamic* store.

We consider this point an interesting one for future investigation.

Towards Class-based Logics. Finally, we think it will be instructive to derive a class-based logic by translating classes into objects, as indicated in Section 3.4. Note that this would automatically provide *modular* reasoning for classes.

Indeed, the encoding from Section 3.4 gives rise to a proof rule

$$\frac{\begin{array}{c} A \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B'_j{::}T'_j\right]_{i\in I', j\in J'} \\ B \equiv \mathsf{Object}(X)\left[\mathsf{f}_i{:}A_i,\ \mathsf{m}_j{:}\varsigma(y_j)B_j{::}T_j\right]_{i\in I\uplus I', j\in J\uplus J'} \\ \Gamma \rhd c : \mathsf{Class}(A){::}T_{\mathsf{res}}(c) \qquad \Gamma \rhd B \prec: A \qquad K \subseteq J' \\ \Gamma \rhd B_j[B/X] \prec: B'_j[B/X] \qquad \vdash_{\mathsf{fo}} T_j \to T'_j \qquad \forall j \in J' - K \\ \Gamma, x{:}B \rhd b_l : B_l[B/X]{::}T_l \qquad \forall l \in J \cup K \end{array}}{\begin{array}{ll} \Gamma \rhd & \mathsf{subclass\ of}\ c{:}\mathsf{Class}(A)\ \mathsf{with}\ (x{:}B) \qquad\qquad : \mathsf{Class}(B) \\ & (\mathsf{f}_i)_{i\in I}\ (\mathsf{m}_j = b_j)_{j\in J}\ (\mathsf{override}\ \mathsf{m}_k = b_k)_{k\in K} \\ & \mathsf{end} \end{array}}$$

similar to the typing rule for classes in Table 3.14. In future work we plan to investigate if this setting provides useful reasoning principles for languages with dynamic class loading and inner classes.

7.3 Summary and Conclusions

In this part of the thesis, based on the denotational semantics of Chapter 3, 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 with respect to an operational semantics, our techniques allowed us to distinguish the notions of derivability and validity: This result was captured in Theorem 5.2.6. Further, we used the denotational framework to extend the logic to recursive object specifications, the corresponding soundness result is Theorem 6.3.9. In comparison to a similar logic presented in (Leino 1998) 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 context Γ cannot reflect the dynamic aspect of the store (which is growing) one uses store specifications Σ. While these store specifications do not show up in the rules of Abadi-Leino logic, they are necessarily preserved by programs, due to the design of the logic. This is shown as part of the soundness proof rather than being a proof obligation on the level of derivations.

By contrast to (Abadi and Leino 2004), we can view store specifications as predicates on stores. However these predicates 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 neither subsumption nor weakening. This led us to a *positive recursive* semantics $[\![A]\!]$ of individual objects, for which the set containment models the syntactic subspecification relation; this is the content of Lemma 4.3.6.

Conditions (M1) – (M3) in the semantics of store specifications (Definitions and Lemmas 5.1.7 and 5.1.8, and 6.2.3 and 6.2.4, respectively) ensure that methods in the store preserve not only the current store specification but also arbitrary extensions $\Sigma' \succcurlyeq \Sigma$. This accounts for the (specifications of) objects allocated inbetween definition time of the method and call time.

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 $\sigma \in [\![\Sigma]\!]$ (cf. Definitions 5.1.7 and 6.2.4 on pages 82 and 107, respectively). More annoyingly, field update requires subspecifications to be invariant in the field components, otherwise even type soundness is invalidated. 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 syntactic subspecification relation instead.

The proof of Theorem 5.1.8, 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: Semantically, a sufficient condition is that transition relations are upwards and downwards closed in their first and second store argument, respectively.

In Section 7.1 we have set our model in the wider context of possible worlds models for languages with store. In Section 7.2 we have also described some of the limitations of Abadi and Leino's logic and sketched potential improvements.

Part III

Reasoning in a Typed Model

Chapter 8

A Typed Semantics for Languages with General References and Subtyping

---

The goal of this part of the thesis is to find a *typed model* of objects. Rather than constructing such a model directly, we proceed by extending and combining several techniques from the literature, building on work of Reddy, Reynolds and Levy.

A call-by-value higher-order language is considered first, with higher-order functions, records, references to values of arbitrary type, and subtyping. We then use the fixed-point encoding of (Reddy 1988; Kamin and Reddy 1994; also see Section 3.2) to obtain a typed denotational semantics of imperative object-oriented languages, both class-based and object-based ones.

An intrinsically typed denotational model for a similar higher-order language but *without* subtyping, based on a possible-world semantics, was recently given by Levy (2002). Here, this model is adapted and related to an untyped model by a logical relation. Following the methodology of Reynolds (2002b), this relation is used to establish coherence of the typed semantics, with a coercion interpretation of subtyping.

8.1 Introduction

Languages such as Standard ML (Milner, Tofte, Harper, and MacQueen 1997) and Scheme (Abelson et al. 1998) allow us to store values of arbitrary types, including function types. Essentially the same effect is pervasive in object-based languages where objects are created on-the-fly and arbitrary method code needs to be kept in the store, as argued in Chapter 3. As explained before, this feature – often referred to as higher-order store or general references – complicates the semantics and logics of such languages considerably (Reus 2002): Besides introducing recursion to the language (Landin 1964), higher-order store in fact requires the semantic domain to be defined by a *mixed-variant* recursive equation.

In fact, only few models of typed languages with general references have appeared in the literature so far (Abramsky, Honda, and McCusker 1998; Ahmed, Appel, and Virga 2002; Levy 2002). Most of the work done on semantics of storage does not readily apply

to languages with higher-order store (Tennent and Ghica 2000).

In a recent paper, Paul Levy proposed a typed semantics for a language with higher-order functions and higher-order store (Levy 2002; Levy 2004). This is a possible worlds model, explicating the dynamic allocation of new (typed) storage locations in the course of a computation. In this chapter, we extend his model to accommodate subtyping by using coercion maps. In the terminology of (Reynolds 2002b), we obtain an *intrinsic* semantics: Meaning is given to derivations of typing judgements, rather than to terms, with the consequence that

- ill-typed phrases are meaningless,
- terms satisfying several judgements will be assigned several meanings, and
- coherence between the meaning of several derivations of the *same* judgement must be established.

Due to the addition of subtyping to Levy's model, derivations are indeed no longer unique and we must prove coherence. A standard approach for such proofs is to transform derivations into a normal form while preserving their semantics. This can be quite involved, even for purely functional languages. For an illustration of this method refer to (Breazu-Tannen, Coquand, Gunter, and Scedrov 1991; Mitchell 1996).

In contrast to intrinsic semantics, an *extrinsic* semantics gives meaning to *all* terms. Types and typing judgements are interpreted as, e.g., (admissible) predicates or partial equivalence relations over an untyped model. Usually, the interpretation of subtyping is straightforward in such models. In (Reynolds 2002b), Reynolds uses a logical relation between intrinsic and extrinsic cpo models of a lambda calculus with subtyping (but no state) to prove coherence. The proof essentially relies on the fact that the denotations of all derivations of a judgement $\Gamma \rhd e : A$ are related to the denotation $[\![e]\!]$ of e in the untyped model underlying the extrinsic semantics, via the *basic lemma* of logical relations (for instance, (Mitchell 1996)). A family of retractions between intrinsic and extrinsic semantics is then used to obtain the meaning of $[\![\Gamma \rhd e : A]\!]$ in terms of Γ, $[\![e]\!]$ and A alone, i.e., independent of any particular derivation of the judgement.

The same ideas are applied to obtain a coherence proof for the language considered here. Two modifications have to be made: Firstly, because of the indexing by worlds we use a *Kripke logical relation* (Mitchell and Moggi 1991) to relate intrinsic and extrinsic semantics — this is straightforward. Secondly, due to the mixed-variant recursion forced by the higher-order store we can no longer use induction over the type structure to establish properties of the relations. In fact even the existence of the Kripke logical relation requires a non-trivial proof — we use the framework of Pitts from Section 2.5 to deal with this complication.

We believe the combination of higher-order storage and subtyping to be interesting in its own right. Nevertheless, since our primary interest in this thesis is in semantics and reasoning principles for object-oriented programs, we note that a number of object encodings use a target language similar to the one considered in this chapter (cf. Section 3.2; also Abadi, Cardelli, and Viswanathan 1996; Kamin and Reddy 1994; Boudol 2004).

While we achieve our goal of constructing a model for the object calculus, our attempts to interpret a logic in the style of Abadi and Leino's were, unfortunately, only partially successful. More precisely, here we give a semantics to the object calculus of Abadi and Cardelli (Abadi and Cardelli 1996). This is done using a typed variant of Kamin and Reddy's closure model (Reddy 1988; Kamin and Reddy 1994). To the best of our knowledge this is the first (intrinsically typed) domain-theoretic model of the imperative object calculus. After presenting this semantics, some evidence that the model of this chapter may serve as basis for program logics is provided, by adapting the *semantic* reasoning techniques of (Reus and Streicher 2004) to a typed setting. We then describe the obstacles to devising a sound syntactic proof calculus for this language.

In the preceding part we have given a denotational semantics for a logic of objects due to (Abadi and Leino 2004), where an untyped cpo model was used. Recall that this logic has a built-in notion of invariance which makes it very similar to a type system. The semantic structure of function types (or method types, more precisely) used in Chapter 5 very much resembles that of various possible worlds models for languages with dynamic allocation (Levy 2002; Reddy and Yang 2004; Stark 1998). We compare our semantics with an extrinsic per semantics derived from the Kripke logical relation over the typed model, again following a construction of (Reynolds 2002b).

8.2 Outline of Part III

In summary, our technical contributions in this chapter are

- we present a model of a language that includes general references and subtyping;
- we successfully apply the ideas of (Reynolds 2002b) to prove coherence of the interpretation; and
- we provide the first (intrinsically typed) model of the imperative object calculus of Abadi and Cardelli (Abadi and Cardelli 1996), based on cpos.

In addition, we shed some more light on the "choice function" construction (page 82) used to establish the existence of store specifications in Chapter 5.

The outline of this part is as follows. In the next section, language and type system are introduced. Then, in Sections 8.4 and 8.5, typed and untyped models are presented. The logical relation is defined next, and retractions between types of the intrinsic semantics and the untyped value space are used to prove coherence in Section 8.7. In Section 8.8 both a derived per semantics and the relation to our earlier work on the interpretation of objects are discussed.

Chapter 9 presents the applications of the theory, providing a semantics of objects and therefore also of classes, by the translation summarised in Section 3.4. We provide several small examples, illustrating the specification and verification of a non-trivial programs. Finally, Chapter 10 concludes this part. Besides a summary of the technical results and a discussion about related work, a number of open questions are listed. Besides, some initial ideas for local reasoning (Reynolds 2002a) about the language considered here are presented.

Table 8.1 *Typing*

$$\frac{\Gamma \rhd e : A \quad A \preceq B}{\Gamma \rhd e : B} \qquad \frac{x{:}A \in \Gamma}{\Gamma \rhd x : A}$$

$$\frac{\Gamma \rhd e_1 : B \quad \Gamma, x{:}B \rhd e_2 : A}{\Gamma \rhd \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : A} \qquad \frac{}{\Gamma \rhd \mathsf{true} : \mathsf{bool}}$$

$$\frac{\Gamma \rhd x : \mathsf{bool} \quad \Gamma \rhd e_1 : A \quad \Gamma \rhd e_2 : A}{\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A} \qquad \frac{}{\Gamma \rhd \mathsf{false} : \mathsf{bool}}$$

$$\frac{\Gamma \rhd x_i : A_i \quad \forall i \in I}{\Gamma \rhd \{\mathsf{m}_i = x_i\}_{i\in I} : \{\mathsf{m}_i : A_i\}_{i\in I}} \qquad \frac{\Gamma \rhd x : \{\mathsf{m}_i : A_i\}_{i\in I}}{\Gamma \rhd x.\mathsf{m}_j : A_j}\ (j \in I)$$

$$\frac{\Gamma, x{:}A \rhd e : B}{\Gamma \rhd \lambda x.e : A \Rightarrow B} \qquad \frac{\Gamma \rhd x : A \Rightarrow B \quad \Gamma \rhd y : A}{\Gamma \rhd x(y) : B}$$

$$\frac{\Gamma \rhd x : A}{\Gamma \rhd \mathsf{new}_A\ x : \mathsf{ref}\ A} \qquad \frac{\Gamma \rhd x : \mathsf{ref}\ A}{\Gamma \rhd \mathsf{deref}\ x : A}$$

$$\frac{\Gamma \rhd x : \mathsf{ref}\ A \quad \Gamma \rhd y : A}{\Gamma \rhd x{:=}y : \mathbf{1}}$$

8.3 Language

Let $\mathbb{L}$ be a countably infinite set of *labels*, ranged over by m. We consider a single base type of booleans, bool, records $\{\mathsf{m}_i : A_i\}_{i\in I}$ with finite index set I, and function types $A \Rightarrow B$. We set $\mathbf{1} \stackrel{def}{=} \{\}$ for the (singleton) type of empty records. Finally, we have a type $\mathsf{ref}\ A$ of mutable references to values of type A. Term forms include lambda calculus terms as well as constructs for creating, dereferencing and updating storage locations. The syntax of types and terms is given by the grammar:

$$\begin{aligned} A, B \in \mathit{Type} &::= \mathsf{bool} \mid \{\mathsf{m}_i : A_i\}_{i\in I} \mid A \Rightarrow B \mid \mathsf{ref}\ A \\ v \in \mathit{Val} &::= x \mid \mathsf{true} \mid \mathsf{false} \mid \{\mathsf{m}_i = x_i\}_{i\in I} \mid \lambda x.e \\ e \in \mathit{Exp} &::= v \mid \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid x.\mathsf{m} \mid x(y) \\ &\quad \mid \mathsf{new}_A\ x \mid \mathsf{deref}\ x \mid x{:=}y \end{aligned}$$

As in our presentation of the object calculus earlier, subterms in most of these term forms are restricted to variables in order to simplify the statement of the semantics in the next section: There, we can exploit the fact that "evaluation" of subterms that exhibit side-effects only appear in the let-construct. However, in subsequent examples we will use a more generous syntax. The reduction of such syntax sugar to the expressions above should always be immediate.

The subtyping relation $A \preceq B$ is the least reflexive and transitive relation closed under the rules

$$\frac{A_i \preceq A'_i \quad \forall i \in I' \quad I' \subseteq I}{\{\mathsf{m}_i : A_i\}_{i\in I} \preceq \{\mathsf{m}_i : A'_i\}_{i\in I'}} \qquad \frac{A' \preceq A \quad B \preceq B'}{A \Rightarrow B \preceq A' \Rightarrow B'}$$

Note that there is no rule for reference types as these need to be invariant, i.e., $\mathsf{ref}\ A \preceq \mathsf{ref}\ B$ holds only if $A \equiv B$. A type inference system is given in Table 8.1, where contexts Γ are finite sets of variable-type pairs, with each variable occurring at most once. As

usual, in writing $\Gamma, x{:}A$ we assume x does not occur in Γ. A subsumption rule is used for subtyping of terms.

8.4 Intrinsic Semantics

In this section we recall the possible worlds model of (Levy 2002). Its extension with records is straightforward, and we interpret the subsumption rule using coercion maps.

8.4.1 Worlds

For each $A \in \mathit{Type}$ let Loc_A be mutually disjoint, countably infinite sets of *locations*. We let l range over $\mathsf{Loc} \stackrel{def}{=} \bigcup_{A \in \mathit{Type}} \mathsf{Loc}_A$, and may use the notation l_A to emphasize that $l \in \mathsf{Loc}_A$. A *world* w is a finite set of locations $l_A \in \mathsf{Loc}$. A world w' *extends* w, written $w' \geq w$, if $w' \supseteq w$. We write $\mathcal{W} = (\mathcal{W}, \leq)$ for the poset of worlds.

8.4.2 Semantic Domain

Recall from Chapter 2 that **pCpo** is the category of cpos (not necessarily containing a least element) and partial continuous functions, and that **Cpo** is the wide subcategory of **pCpo** where morphisms are total.

Informally, a world describes the shape of the store, i.e., the number and names of locations of each type allocated in the store. In the semantics we want a cpo S_w of w-stores for each $w \in \mathcal{W}$, and a cpo $[\![A]\!]_w$ of values of type A. In fact, we require that each $[\![A]\!]$ denotes a co-variant functor from $\mathcal{W}$ (considered as posetal category) to **Cpo**, formalising the intuition that values can always be used with larger stores. We write the image of the morphism $w \leq w'$ under $[\![A]\!]$ as $[\![A]\!]_w^{w'}$. Note that, in contrast to types, the store S is *not* assumed to be (either co- or contra-variant) functorial: In general there is no obvious way to either enlarge or restrict a store (Levy 2004).

The cpo of w-stores is defined as

$$S_w \stackrel{def}{=} \{\!|\, l_A : [\![A]\!]_w \,|\!\}_{l_A \in w} \tag{8.1}$$

consisting of records that provide a value of type A for every location l_A listed in the world w. For worlds $w \in \mathcal{W}$, $[\![\mathsf{bool}]\!]_w \stackrel{def}{=} \mathsf{BVal}$ denotes the set $\{\mathit{true}, \mathit{false}\}$ of truth values considered as discrete cpo, and similarly $[\![\mathsf{ref}\ A]\!]_w \stackrel{def}{=} \{l_A \mid l_A \in w\}$ is the discretely ordered cpo of A-locations allocated in w-stores. Further, $[\![\{\mathsf{m}_i : A_i\}_{i \in I}]\!]_w \stackrel{def}{=} \{\!|\, \mathsf{m}_i : [\![A_i]\!]_w \,|\!\}$ is the cpo of all records $\{\!|\, \mathsf{m}_i = a_i \,|\!\}_{i \in I}$ with component m_i in $[\![A_i]\!]_w$, ordered pointwise. On morphisms $w \leq w'$, $[\![\mathsf{bool}]\!]_w^{w'} \stackrel{def}{=} \mathsf{id}_{\mathsf{BVal}}$ is the identity map, and $[\![\mathsf{ref}\ A]\!]_w^{w'}$ is the inclusion $[\![\mathsf{ref}\ A]\!]_w \subseteq [\![\mathsf{ref}\ A]\!]_{w'}$. For records, $[\![\{\mathsf{m}_i : A_i\}]\!]_w^{w'} \stackrel{def}{=} \lambda r.\{\!|\, \mathsf{m}_i = [\![A_i]\!]_w^{w'}\ (r.\mathsf{m}_i) \,|\!\}$ acts pointwise on the components. The type of functions $A \Rightarrow B$ is the most interesting, since it involves the store S:

$$[\![A \Rightarrow B]\!]_w \stackrel{def}{=} \textstyle\prod_{w' \geq w} (S_{w'} \times [\![A]\!]_{w'} \rightharpoonup \sum_{w'' \geq w'} (S_{w''} \times [\![B]\!]_{w''})) \tag{8.2}$$

This says that a function $f \in [\![A \Rightarrow B]\!]_w$ may be applied in any future (larger) store w' to a w'-store s and value $v \in [\![A]\!]_{w'}$. The computation $f_{w'}(s, v)$ may allocate new storage,

Table 8.2 *Semantics of types*

$[\![\mathsf{bool}]\!]_w$	$\stackrel{def}{=}$	BVal
$[\![\mathsf{bool}]\!]_w^{w'}$	$\stackrel{def}{=}$	$\mathsf{id}_{\mathsf{BVal}}$
$[\![\mathsf{ref}\ A]\!]_w$	$\stackrel{def}{=}$	$\{l_A \mid l_A \in w\}$
$[\![\mathsf{ref}\ A]\!]_w^{w'}$	$\stackrel{def}{=}$	$\lambda l.l$
$[\![\{\mathsf{m}_i : A_i\}_{i\in I}]\!]_w$	$\stackrel{def}{=}$	$\{\!\!\{\mathsf{m}_i : [\![A_i]\!]_i \}\!\!\}_{i\in I}$
$[\![\{\mathsf{m}_i : A_i\}_{i\in I}]\!]_w^{w'}$	$\stackrel{def}{=}$	$\lambda r.\{\!\!\{\mathsf{m}_i = [\![A_i]\!]_w^{w'}\ (r.\mathsf{m}_i)\}\!\!\}_{i\in I}$
$[\![A \Rightarrow B]\!]_w$	$\stackrel{def}{=}$	$\prod_{w'\geq w}(S_{w'} \times [\![A]\!]_{w'} \rightharpoonup \sum_{w''\geq w'}(S_{w''} \times [\![B]\!]_{w''}))$
$[\![A \Rightarrow B]\!]_w^{w'}$	$\stackrel{def}{=}$	$\lambda f\, \lambda_{w''\geq w'}.f_{w''}$
S_w	$\stackrel{def}{=}$	$\{\!\!\{l_A : [\![A]\!]_w \}\!\!\}_{l_A\in w}$

and upon termination it yields a store and value in a yet larger world $w'' \geq w'$. For a morphism $w \leq w'$, $[\![A \Rightarrow B]\!]_w^{w'}(f) = \lambda_{w''\geq w'} f_{w''}$ is the restriction to worlds $w'' \geq w'$. Table 8.2 summarises these requirements.

Equations (8.1) and (8.2) clearly show the effect of allowing higher order store: Since functions $A \Rightarrow B$ can also be stored, S and $[\![A \Rightarrow B]\!]$ are mutually recursive. Due to the use of S in both positive and negative positions in (8.2) a mixed-variant domain equation for S must be solved. To this end, in (Levy 2002) a bilimit-compact category $\mathcal{C}$ is considered in which the above semantic requirements can be interpreted. From bilimit-compactness it follows that every locally continuous functor $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ has a minimal invariant, i.e., an object D in $\mathcal{C}$ such that $F(D,D) = D$ (omitting isomorphisms) and id_D is the least fixed point of the continuous endofunction $\delta : \mathcal{C}(D,D) \to \mathcal{C}(D,D)$ given by $\delta(e) = F(e,e)$ (cf. Chapter 2).

Following (Levy 2002), the semantics of types can thus be obtained as minimal invariant of the locally continuous functor $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ that is derived from the domain equations for types (by separating positive and negative occurrences of the store), and given in Table 8.3. Here, $\mathcal{C}$ is the bilimit-compact category

$$\mathcal{C} \stackrel{def}{=} \prod_{w\in\mathcal{W}} \mathbf{pCpo} \times \prod_{A\in Type}[\mathcal{W}, \mathbf{Cpo}] \bullet\!\!\rightarrow [\mathcal{W}, \mathbf{pCpo}] \tag{8.3}$$

Recall from Chapter 2 that by $[\mathcal{W}, \mathbf{Cpo}] \bullet\!\!\rightarrow [\mathcal{W}, \mathbf{pCpo}]$ we denote the category where objects are functors $A, B : \mathcal{W} \to \mathbf{Cpo}$ and morphisms are partial natural transformations $\mu : A \dot{\to} B$, i.e., for $A, B : \mathcal{W} \to \mathbf{Cpo}$ the diagram

$$\begin{array}{ccc} A_w & \xrightarrow{\mu_w} & B_w \\ {\scriptstyle A_w^{w'}}\downarrow & & \downarrow{\scriptstyle B_w^{w'}} \\ A_{w'} & \xrightarrow[\mu_{w'}]{} & B_{w'} \end{array} \tag{8.4}$$

commutes. Thus, the objects of $\mathcal{C}$ are pairs $D = \langle \{D_{Sw}\}_w, \{D_A\}_A \rangle$ consisting of a $\mathcal{W}$-indexed family of cpos D_{Sw}, and a *Type*-indexed family of functors $D_A : \mathcal{W} \longrightarrow \mathbf{Cpo}$. A

Table 8.3 *Functor* $F : C^{op} \times C \longrightarrow C$

On C-objects D, E

$$
\begin{array}{lcl}
F(D,E)_{Sw} & = & \{\!|\, l_A : E_{Aw} \,|\!\}_{l_A \in w} \\
F(D,E)_{\mathsf{bool}w} & = & \mathsf{BVal} = \{true, false\} \\
F(D,E)_{\mathsf{bool}(w \leq w')} & = & \mathsf{id}_{\mathsf{BVal}} \\
F(D,E)_{\{\mathsf{m}_i:A_i\}w} & = & \{\!|\, \mathsf{m}_i : E_{A_i w} \,|\!\} \\
F(D,E)_{\{\mathsf{m}_i:A_i\}(w \leq w')} & = & \lambda r.\{\!|\, \mathsf{m}_i = E_{A_i(w \leq w')}(r.\mathsf{m}_i) \,|\!\} \\
F(D,E)_{A \Rightarrow Bw} & = & \prod_{w' \geq w}(D_{Sw'} \times D_{Aw'} \rightharpoonup \sum_{w'' \geq w'}(E_{Sw''} \times E_{Bw''})) \\
F(D,E)_{A \Rightarrow B(w \leq w')} & = & \lambda f \lambda w'' \geq w'.f_{w''} \\
F(D,E)_{\mathsf{ref}\, Aw} & = & \{l_A \mid l_A \in w\} \\
F(D,E)_{\mathsf{ref}\, A(w \leq w')} & = & \lambda l.l
\end{array}
$$

On C-morphisms $h : D' \longrightarrow D$ and $k : E \longrightarrow E'$ by

$$
\begin{array}{lcl}
F(h,k)_{Sw} & = & \lambda s.\begin{cases} l_A \mapsto k_{Sw}(s)_{l_A} & \text{if } k_{Sw}(s)_{l_A}\downarrow \text{ for all } l_A \in w \\ \text{undefined} & \text{otherwise} \end{cases} \\
F(h,k)_{\mathsf{bool}w} & = & \mathsf{id}_{\mathsf{BVal}} \\
F(h,k)_{\{\mathsf{m}_i:A_i\}w} & = & \lambda r.\begin{cases} \{\!|\, \mathsf{m}_i = k_{A_i w}(r.\mathsf{m}_i) \,|\!\} & \text{if } k_{A_i w}(r.\mathsf{m}_i)\downarrow \text{ for all } i \\ \text{undefined} & \text{otherwise} \end{cases} \\
F(h,k)_{A \Rightarrow Bw} & = & \lambda f \lambda w' \geq w\, \lambda\langle s, a\rangle. \\
 & & \quad \begin{cases} \langle w'', \langle k_{Sw''}(s''), k_{Bw''}(b)\rangle\rangle & \\ \qquad \text{if } h_{Sw'}(s)\downarrow \text{ and } h_{Aw'}(a)\downarrow \text{ and} & \\ \qquad f_{w'}(h_{Sw'}(s), h_{Aw'}(a)) = \langle w'', \langle s'', b\rangle\rangle & \\ \qquad \text{and } k_{Sw''}(s'')\downarrow \text{ and } k_{Bw''}(b)\downarrow & \\ \text{undef.} \quad \text{otherwise} & \end{cases} \\
F(h,k)_{\mathsf{ref}\, Aw} & = & \lambda l.l
\end{array}
$$

morphism $h \in C(D,E)$ consists of partial continuous maps $h_{Sw} : D_{Sw} \rightharpoonup E_{Sw}$ for each $w \in \mathcal{W}$, and partial natural transformations $h_A : D_A \stackrel{\cdot}{\rightarrow} E_A$, for each $A \in \mathit{Type}$.

The first component of the product in (8.3) is used to obtain the recursively defined cpos $S_w \stackrel{def}{=} D_{Sw}$ of stores from the minimal invariant $D = \langle \{D_{Sw}\}_w, \{D_A\}_A\rangle$. The second component yields $[\![A]\!] \stackrel{def}{=} D_A$ as the denotation of types. In fact, since $[\mathcal{W}, \mathbf{Cpo}]$ is sub-bilimit-compact within $[\mathcal{W}, \mathbf{Cpo}] \bullet\!\!\rightarrow [\mathcal{W}, \mathbf{pCpo}]$ by Proposition 2.4.1, the minimal invariant D provides for every $A \in \mathit{Type}$ an isomorphism $F(D,D)_A = D_A$ in the category $[\mathcal{W}, \mathbf{Cpo}]$. Conceptually, therefore, the model exhibits the common structure of call-by-value models: the interpretation of a value $x_1{:}A_1, \ldots, x_n{:}A_n \rhd v : A$ can be given by a total natural transformation from $[\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$ to $[\![A]\!]$ where the cpos $[\![A]\!]_w$ need not necessarily be pointed. We refer to (Levy 2004) for more details.

Table 8.4 *Coercion maps*

$$\left[\!\left[\frac{}{A \preceq A}\right]\!\right]_w = \mathsf{id}_{[\![A]\!]_w}$$

$$\left[\!\left[\frac{\mathcal{P}(A \preceq A') \quad \mathcal{P}(A' \preceq B)}{A \preceq B}\right]\!\right]_w = [\![\mathcal{P}(A' \preceq B)]\!]_w \circ [\![\mathcal{P}(A \preceq A')]\!]_w$$

$$\left[\!\left[\frac{I' \subseteq I \quad \mathcal{P}(A_i \preceq A'_i)\ \forall i \in I'}{\{\mathsf{m}_i : A_i\}_{i\in I} \preceq \{\mathsf{m}_i : A'_i\}_{i\in I'}}\right]\!\right]_w = \lambda r.\{\!|\mathsf{m}_i = [\![\mathcal{P}(A_i \preceq A'_i)]\!]_w\,(r.\mathsf{m}_i)|\!\}_{i\in I'}$$

$$\left[\!\left[\frac{\mathcal{P}(A' \preceq A) \quad \mathcal{P}(B \preceq B')}{A \Rightarrow B \preceq A' \Rightarrow B'}\right]\!\right]_w$$

$$= \lambda f \lambda_{w'\geq w} \lambda\langle s, x\rangle. \begin{cases} \langle w'', \langle s', [\![\mathcal{P}(B \preceq B')]\!]_{w''}\, x'\rangle\rangle \\ \qquad\qquad \text{if } f_{w'}\langle s, [\![\mathcal{P}(A' \preceq A)]\!]_{w'}\,(x)\rangle = \langle w'', \langle s', x'\rangle\rangle \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

8.4.3 Semantics

Each subtyping derivation $A \preceq B$ determines a *coercion*, which is in fact a (total) natural transformation from $[\![A]\!]$ to $[\![B]\!]$, defined in Table 8.4: We follow the notation of (Reynolds 2002b) and write $\mathcal{P}(J)$ to distinguish a *derivation* of judgement J from the judgement itself.

In the following we write $[\![\Gamma]\!]_w$ for the set of environments, i.e., maps from variables $\mathsf{dom}(\Gamma)$ to $\bigcup_A [\![A]\!]_w$ such that $\rho(x) \in [\![A]\!]_w$ for all $x{:}A \in \Gamma$. For $w \preceq w'$, $[\![\Gamma]\!]_w^{w'}(\rho)$ denotes the environment such that $[\![\Gamma]\!]_w^{w'}(\rho)(x) = [\![A]\!]_w^{w'}(\rho(x))$ for $x{:}A$ in Γ. The semantics of (derivations of) typing judgments can now be presented,

$$[\![\Gamma \rhd e : A]\!]_w : [\![\Gamma]\!]_w \to S_w \to \textstyle\sum_{w'\geq w}(S_{w'} \times [\![A]\!]_{w'})$$

Remark 8.4.1. *As observed in Levy's paper, each* value $\Gamma \rhd v : A$ *determines a natural transformation from* $[\![\Gamma]\!]$ *to* $[\![A]\!]$ *in* $[\mathcal{W}, \mathbf{Cpo}]$. *In our case this is a consequence of the fact that (i) values do not affect the store and (ii) coercion maps determine (total) natural transformations. We make use of this fact in the definition of the semantics. For example, in the case of records we do not have to fix an order for the evaluation of the components.*

The semantics of subtyping judgements is used for the interpretation of the subsumption rule,

$$\left[\!\left[\frac{\mathcal{P}(\Gamma \rhd e : A) \quad \mathcal{P}(A \preceq B)}{\Gamma \rhd e : B}\right]\!\right]_w \rho s$$

$$= \begin{cases} \langle w', \langle s', [\![\mathcal{P}(A \preceq B)]\!]_{w'}\, a\rangle\rangle & \text{if } [\![\mathcal{P}(\Gamma \rhd e : A)]\!]_w\, \rho s = \langle w', \langle s', a\rangle\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

As explained above, the semantics of functions is parameterised over extensions of the current world w,

$$\left[\!\left[\frac{\mathcal{P}(\Gamma, x : A \rhd e : B)}{\Gamma \rhd \lambda x.e : A \Rightarrow B}\right]\!\right]_w \rho s$$

$$= \langle w, \langle s, \lambda w' \geq w \lambda\langle s', a\rangle.\, [\![\mathcal{P}(\Gamma, x{:}A \rhd e : B)]\!]_{w'}\, ([\![\Gamma]\!]_w^{w'}\, \rho)[x := a]\, s'\rangle\rangle$$

Table 8.5 *Semantics of typing judgements*

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd e : A) \quad \mathcal{P}(A \preceq B)}{\Gamma \rhd e : B} \right]\!\!\right]_w \rho s = \begin{cases} \langle w', \langle s', [\![\mathcal{P}(A \preceq B)]\!]_{w'}\, a\rangle\rangle & \text{if } [\![\mathcal{P}(\Gamma \rhd e : A)]\!]_w\, \rho s = \langle w', \langle s', a\rangle\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\frac{}{\Gamma \rhd x : A} \right]\!\!\right]_w \rho s = \langle w, \langle s, \rho(x)\rangle\rangle$$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd e_1 : B) \quad \mathcal{P}(\Gamma, x{:}B \rhd e_2 : A)}{\Gamma \rhd \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : A} \right]\!\!\right]_w \rho s = \begin{cases} [\![\mathcal{P}(\Gamma, x{:}B \rhd e_2 : A)]\!]_{w'}\, ([\![\Gamma]\!]_w^{w'}\, \rho)[x := b]\, s' \\ \qquad \text{if } [\![\mathcal{P}(\Gamma \rhd e_1 : B)]\!]_w\, \rho s = \langle w', \langle s', b\rangle\rangle \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\frac{}{\Gamma \rhd \mathsf{true} : \mathsf{bool}} \right]\!\!\right]_w \rho s = \langle w, \langle s, \mathit{true}\rangle\rangle$$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : \mathsf{bool}) \quad \mathcal{P}(\Gamma \rhd e_i : A) \quad i = 1, 2}{\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A} \right]\!\!\right]_w \rho s = \begin{cases} [\![\mathcal{P}(\Gamma \rhd e_1 : A)]\!]_w\, \rho s & \text{if } [\![\mathcal{P}(\Gamma \rhd x : \mathsf{bool})]\!]_w\, \rho s = \langle w, \langle s, \mathit{true}\rangle\rangle \\ [\![\mathcal{P}(\Gamma \rhd e_2 : A)]\!]_w\, \rho s & \text{if } [\![\mathcal{P}(\Gamma \rhd x : \mathsf{bool})]\!]_w\, \rho s = \langle w, \langle s, \mathit{false}\rangle\rangle \end{cases}$$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x_i : A_i) \quad \forall i \in I}{\Gamma \rhd \{\mathsf{m}_i = x_i\}_{i\in I} : \{\mathsf{m}_i : A_i\}_{i\in I}} \right]\!\!\right]_w \rho s = \langle w, \langle s, \{\!|\mathsf{m}_i = a_i|\!\}_{i\in I}\rangle\rangle$$

where $\langle w, \langle s, a_i\rangle\rangle = [\![\mathcal{P}(\Gamma \rhd x_i : A_i)]\!]_w\, \rho s$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : \{\mathsf{m}_i : A_i\}_{i\in I})}{\Gamma \rhd x.\mathsf{m}_i : A_i} \right]\!\!\right]_w \rho s = \langle w, \langle s, a.\mathsf{m}\rangle\rangle$$

where $\langle w, \langle s, a\rangle\rangle = [\![\mathcal{P}(\Gamma \rhd x : \{\mathsf{m}_i : A_i\})]\!]_w\, \rho s$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma, x : A \rhd e : B)}{\Gamma \rhd \lambda x.e : A \Rightarrow B} \right]\!\!\right]_w \rho s = \langle w, \langle s, \lambda w' \geq w \lambda\langle s', a\rangle.\, [\![\mathcal{P}(\Gamma, x{:}A \rhd e : B)]\!]_{w'}\, ([\![\Gamma]\!]_w^{w'}\, \rho)[x := a]\, s'\rangle\rangle$$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : A \Rightarrow B) \quad \mathcal{P}(\Gamma \rhd y : A)}{\Gamma \rhd x(y) : B} \right]\!\!\right]_w \rho s = f_w(s, a)$$

where $\langle w, \langle s, f\rangle\rangle = [\![\mathcal{P}(\Gamma \rhd x : A \Rightarrow B)]\!]_w\, \rho s$
and $\langle w, \langle s, a\rangle\rangle = [\![\mathcal{P}(\Gamma \rhd y : A)]\!]_w\, \rho s$

Table 8.6 *Semantics of typing judgements (continued)*

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : A)}{\Gamma \rhd \mathsf{new}_A\ x : \mathsf{ref}\ A} \right]\!\!\right]_w \rho s = \langle w', \langle s', l_A \rangle \rangle$$

where $\langle w, \langle s, a \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd x : A)]\!]_w\, \rho s$,
$w' = w \cup \{l_A\}$ for $l_A \in \mathsf{Loc}_A \setminus \mathsf{dom}(w)$ and for all $l' \in w'$:

$$s'.l' = \begin{cases} [\![A']\!]_w^{w'}\ (s.l') & \text{for } l' \in w \cap \mathsf{Loc}_{A'} \\ [\![A]\!]_w^{w'}\ (a) & \text{for } l' = l_A \end{cases}$$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : \mathsf{ref}\ A)}{\Gamma \rhd \mathsf{deref}\ x : A} \right]\!\!\right]_w \rho s = \langle w, \langle s, s.l \rangle \rangle$$

where $\langle w, \langle s, l \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd x : \mathsf{ref}\ A)]\!]_w\, \rho s$

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : \mathsf{ref}\ A) \quad \mathcal{P}(\Gamma \rhd y : A)}{\Gamma \rhd x{:=}y : \mathbf{1}} \right]\!\!\right]_w \rho s = \langle w, \langle \hat{s}, \{\!|\,|\!\} \rangle \rangle$$

where $\langle w, \langle s, l \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd x : \mathsf{ref}\ A)]\!]_w\, \rho s$,
$\langle w, \langle s, a \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd y : A)]\!]_w\, \rho s$ and for $l' \in w$:

$$\hat{s}.l' = \begin{cases} a & \text{if } l' = l \\ s.l' & \text{if } l' \neq l \end{cases}$$

Function application is

$$\left[\!\!\left[\frac{\mathcal{P}(\Gamma \rhd x : A \Rightarrow B) \quad \mathcal{P}(\Gamma \rhd y : A)}{\Gamma \rhd x(y) : B} \right]\!\!\right]_w \rho s = f_w(s, a)$$

where the premiss of the rule yields $\langle w, \langle s, f \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd x : A \Rightarrow B)]\!]_w\, \rho s$ and $\langle w, \langle s, a \rangle \rangle = [\![\mathcal{P}(\Gamma \rhd y : A)]\!]_w\, \rho s$. Most of the remaining cases are similarly straightforward; Tables 8.5 and 8.6 contain the complete definition.

8.5 An Untyped Semantics

We also give an *untyped* semantics of the language in the (bilimit-compact) category **pCpo**. Let Val be a cpo satisfying

$$\mathsf{Val} \quad = \quad \mathsf{BVal} + \mathsf{Loc} + \mathsf{Rec}_{\mathbb{L}}(\mathsf{Val}) + (\mathsf{St} \times \mathsf{Val} \rightharpoonup \mathsf{St} \times \mathsf{Val}) \tag{8.5}$$

where $\mathsf{St} \stackrel{def}{=} \mathsf{Rec}_{\mathsf{Loc}}(\mathsf{Val})$ denotes the cpo of untyped records with labels from Loc, ordered by $r_1 \sqsubseteq r_2$ iff $\mathsf{dom}(r_1) = \mathsf{dom}(r_2)$ and $r_1.\mathsf{m} \sqsubseteq r_2.\mathsf{m}$ for all $\mathsf{m} \in \mathsf{dom}(r_1)$; see Chapter 2. The interpretation of terms, $[\![e]\!] : \mathsf{Env} \to \mathsf{St} \rightharpoonup \mathsf{St} \times \mathsf{Val}$, is essentially straightforward, typical cases are those of abstraction and application:

$$[\![\lambda x.e]\!]\, \eta\sigma = \langle \sigma, \lambda\langle \sigma', v \rangle.\ [\![e]\!]\, \eta[x := v]\, \sigma' \rangle$$

$$[\![x(y)]\!]\, \eta\sigma = \begin{cases} \eta(x)\langle \sigma, \eta(y) \rangle & \text{if } \eta(x) \in [\mathsf{St} \times \mathsf{Val} \rightharpoonup \mathsf{St} \times \mathsf{Val}] \text{ and } \eta(y)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Compared to the intrinsic semantics of the previous section, there are now many more possibilities of undefinedness if things "go wrong", for instance, if the denotation of x in $x(y)$ is not a function value.

The semantics of new_A may be slightly surprising as there is still some type information in the choice of locations: If $\eta(x)\downarrow$, define

$$[\![\mathsf{new}_A\ x]\!]\,\eta\sigma = \langle \sigma + \{\!|l_A = \eta(x)|\!\}, l_A\rangle \qquad \text{where } l_A \in \mathsf{Loc}_A \setminus \mathsf{dom}(\sigma)$$

and undefined otherwise. Informally, the worlds of the intrinsic semantics are encoded in the domain of untyped stores. Although σ with $\mathsf{dom}(\sigma) = w$ does not necessarily correspond to a (typed) w-store in any sense, this will be the case for stores being derived from well-typed terms. This is one of the results of Section 8.6 below; see also the discussion in Section 8.8.1.

The remaining cases of the definition are given in Table 8.7.

8.6 A Kripke Logical Relation

In Reynolds' (2002b) article a logical relation between typed and untyped models was used to establish coherence of the typed semantics. Here this must be slightly generalised to a *Kripke logical relation* because of the possible worlds semantics of types. Kripke logical relations have appeared in work on Kripke lambda models (Mitchell and Moggi 1991), and more recently in connection with dynamic name creation in the nu-calculus (Zhang and Nowak 2003; Goubault-Larrecq, Lasota, and Nowak 2002). Kripke logical relations are not only indexed by types but also by possible worlds, subject to a monotonicity condition (Lemma 8.6.5 below) stating that related elements remain in relation when moving from a smaller to a larger world.

In Table 8.8 we now define such a family of *Type*- and $\mathcal{W}$-indexed relations $R^A_w \subseteq [\![A]\!]_w \times \mathsf{Val}$. Note that the existence of this family R is not straightforward: There are both positive and negative occurrences of R^{St}_w in the clause for function types $A \Rightarrow B$. Consequently, R cannot be defined by induction on the type structure, nor does it give rise to a monotonic operation on (the complete lattice of) admissible predicates so that the Tarski fixed point theorems would apply directly.

8.6.1 Existence of R^A_w

To establish the existence of such a relation one uses Pitts' technique for the bilimit-compact product category $C \times \mathbf{pCpo}$. Let $G : \mathbf{pCpo}^{op} \times \mathbf{pCpo} \longrightarrow \mathbf{pCpo}$ be the locally continuous functor for which (8.5) is the minimal invariant,

$$G(D,E) \;=\; \mathsf{BVal} + \mathsf{Loc} + \mathsf{Rec}_{\mathbb{L}}(E) + (\mathsf{Rec}_{\mathsf{Loc}}(D) \times D \multimap \mathsf{Rec}_{\mathsf{Loc}}(E) \times E)$$

and let F be the functor defined in Table 8.3 on page 133. Therefore $\langle D, \mathsf{Val}\rangle$ is the minimal invariant of $F \times G$. A (normal) relational structure $\mathcal{R}$ on the category $C \times \mathbf{pCpo}$, in the sense of Section 2.5, is given by the following data.

- For each object $\langle X, Y\rangle$ of $C \times \mathbf{pCpo}$, let $\mathcal{R}(X,Y)$ consist of the type- and world-indexed families R of admissible relations, where for $A \in \mathit{Type}$ and $w \in \mathcal{W}$, $R^A_w \subseteq X_{Aw} \times Y$ and $R^{St}_w \subseteq X_{Sw} \times \mathsf{Rec}_{\mathsf{Loc}}(Y)$.

Table 8.7 *Untyped interpretation of terms*

$$[\![x]\!]\,\eta\sigma = \begin{cases} \langle \sigma, \eta(x)\rangle & \text{if } \eta(x)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2]\!]\,\eta\sigma = \begin{cases} [\![e_2]\!]\,\eta[x := v]\,\sigma' & \text{if } [\![e_1]\!]\,\eta\sigma = \langle \sigma', v\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{true}]\!]\,\eta\sigma = \langle \sigma, \mathit{true}\rangle$$

$$[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\,\eta\sigma = \begin{cases} [\![e_1]\!]\,\eta\sigma & \text{if } \eta(x) = \mathit{true} \\ [\![e_2]\!]\,\eta\sigma & \text{if } \eta(x) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\{\mathsf{m}_i = x_i\}]\!]\,\eta\sigma = \begin{cases} \langle \sigma, \{\!|\mathsf{m}_i = \eta(x_i)|\!\}\rangle & \text{if } \eta(x_i)\downarrow \text{ for all } i \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![x.\mathsf{m}]\!]\,\eta\sigma = \begin{cases} \langle \sigma, \eta(x).\mathsf{m}\rangle & \text{if } \eta(x) \in \mathsf{Rec}_{\mathcal{M}}(\mathsf{Val}) \text{ and } \eta(x).\mathsf{m}\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\lambda x.a]\!]\,\eta\sigma = \langle \sigma, \lambda\langle \sigma', v\rangle.\ [\![a]\!]\,\eta[x := v]\,\sigma'\rangle$$

$$[\![x(y)]\!]\,\eta\sigma = \begin{cases} \eta(x)\langle \sigma, \eta(y)\rangle & \text{if } \eta(x) \in [\mathsf{St}\times\mathsf{Val} \rightharpoonup \mathsf{St}\times\mathsf{Val}] \\ & \text{and } \eta(y)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{new}_A\ x]\!]\,\eta\sigma = \langle \sigma + \{\!|l_A = \eta(x)|\!\}, l_A\rangle, \text{ where } l_A \in \mathsf{Loc}_A \setminus \mathsf{dom}(\sigma)$$

$$[\![\mathsf{deref}\ x]\!]\,\eta\sigma = \begin{cases} \langle \sigma, \sigma.\eta(x)\rangle & \text{if } \eta(x) \in \mathsf{Loc} \text{ and } \sigma.\eta(x)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![x{:=}y]\!]\,\eta\sigma = \begin{cases} \langle \sigma', \{\!|\,|\!\}\rangle & \text{if } \eta(x) \in \mathsf{Loc}, \sigma.\eta(x)\downarrow \text{ and } \eta(y)\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{where } \sigma'.l = \begin{cases} \eta(y) & \text{if } l = \eta(x) \\ \sigma.l & \text{otherwise} \end{cases}$$

· For morphisms $f = \langle f_1, f_2\rangle : \langle X, Y\rangle \to \langle X', Y'\rangle$, and relations $R \in \mathcal{R}(X, Y)$ and $S \in \mathcal{R}(X', Y')$, we define $\langle f_1, f_2\rangle : R \subset S$ *iff* for all $w \in \mathcal{W}$, $A \in \mathit{Type}$, for all $x \in X_{Aw}$, $y \in Y$, $s \in X_{Sw}$ and $\sigma \in \mathsf{Rec}_{\mathsf{Loc}}(Y)$:

$$\langle x, y\rangle \in R^A_w \implies \begin{cases} f_{1\,Aw}(x)\uparrow \wedge f_2(y)\uparrow \text{ or} \\ f_{1\,Aw}(x)\downarrow \wedge f_2(y)\downarrow \wedge \langle f_{1\,Aw}(x), f_2(y)\rangle \in S^A_w \end{cases}$$

$$\langle s, \sigma\rangle \in R^{St}_w \implies \begin{cases} f_{1\,Sw}(x)\uparrow \wedge \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma)\uparrow \text{ or} \\ f_{1\,Sw}(x)\downarrow \wedge \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma)\downarrow \ \wedge \langle f_{1\,Sw}(x), \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma)\rangle \in S^{St}_w \end{cases}$$

Firstly, it is easy to see that this relational structure $\mathcal{R}$ on $\mathcal{C}\times\mathbf{pCpo}$ indeed satisfies the axioms **(Identity)**, **(Composition)** and **(Normality)** of Section 2.5.1:

Table 8.8 *Kripke logical relation*

$$
\begin{array}{lcl}
\langle x,y\rangle \in R_w^{\mathsf{bool}} & \stackrel{def}{\iff} & y \in \mathsf{BVal} \ \wedge \ x = y \\
\langle r,s\rangle \in R_w^{\{\mathsf{m}_i:A_i\}} & \stackrel{def}{\iff} & s \in \mathsf{Rec}_{\mathbb{L}}(\mathsf{Val}) \ \wedge \ \forall i.\, (s.\mathsf{m}_i\!\downarrow \ \wedge \langle r.\mathsf{m}_i, s.\mathsf{m}_i\rangle \in R_w^{A_i}) \\
\langle f,g\rangle \in R_w^{A\Rightarrow B} & \stackrel{def}{\iff} & g \in [\mathsf{St}\times\mathsf{Val} \rightharpoonup \mathsf{St}\times\mathsf{Val}] \ \wedge \\
 & & \forall w' \geq w \ \forall \langle s,\sigma\rangle \in R_{w'}^{St} \ \forall \langle x,y\rangle \in R_{w'}^{A} \\
 & & \quad (f_{w'}(s,x)\!\uparrow \ \wedge\, g(\sigma,y)\!\uparrow) \\
 & & \vee \ \exists w'' \geq w' \ \exists s' \in S_{w'} \ \exists x' \in [\![B]\!]_{w'} \ \exists \sigma' \in \mathsf{St} \ \exists y' \in \mathsf{Val}. \\
 & & \quad\ (f_{w'}(s,x) = \langle w'', \langle s',x'\rangle\rangle \ \wedge \ g(\sigma,y) = \langle \sigma', y'\rangle \\
 & & \quad\ \wedge \langle s',\sigma'\rangle \in R_{w''}^{St} \ \wedge \ \langle x',y'\rangle \in R_{w''}^{B}) \\
\langle x,y\rangle \in R_w^{\mathsf{ref}\,A} & \stackrel{def}{\iff} & y \in w \cap \mathsf{Loc}_A \ \wedge \ x = y
\end{array}
$$

with the auxiliary relation $R_w^{St} \subseteq S_w \times \mathsf{St}$,

$$
\langle s,\sigma\rangle \in R_w^{St} \quad \stackrel{def}{\iff} \quad \mathsf{dom}(s) = w = \mathsf{dom}(\sigma) \ \wedge \ \forall l_A \in w.\, \langle s.l_A, \sigma.l_A\rangle \in R_w^A
$$

- $\langle \mathsf{id}_X, \mathsf{id}_Y\rangle : R \subset R$, for all objects $\langle X,Y\rangle$ and all $\mathcal{R}$-relations $R \in \mathcal{R}(X,Y)$;
- for composable f,g with $f : S \subset T$ and $g : R \subset S$ we have $f \circ g : R \subset T$; and
- if $\mathsf{id} : R \subset S$ and $\mathsf{id} : S \subset R$ then $R = S$.

Further, for objects $\langle X,Y\rangle$, $\langle X',Y'\rangle$ and relations $R \in \mathcal{R}(X,Y)$ and $S \in \mathcal{R}(X',Y')$, the subset

$$
[R,S] \ \stackrel{def}{=} \ \{\langle f,g\rangle \mid \langle f,g\rangle : R \subset S\}
$$

of $[X \rightharpoonup X'] \times [Y \rightharpoonup Y']$ has a least element and is chain-closed, thus $\mathcal{R}_{adm} = \mathcal{R}$:

- Clearly, the pair $\bot = \langle \bot, \bot\rangle$ of maps that are undefined everywhere satisfies $\bot : R \subset S$ and is below every other $f \in [R,S]$.
- Let $\langle f_0,g_0\rangle \sqsubseteq \langle f_1,g_1\rangle \sqsubseteq \ldots$ in $[R,S]$ and $\langle f,g\rangle = \bigsqcup_i \langle f_i,g_i\rangle$. Now suppose $\langle x,y\rangle \in X_{Aw} \times Y$ s.t. $\langle x,y\rangle \in R_w^A$ and $f_{Aw}(x)\downarrow$ ($g(y)\downarrow$, resp.). Then $f_{iAw}(x)\downarrow$ ($g_i(y)\downarrow$, resp.) for all sufficiently large i, which entails that also $g_i(y)\downarrow$ ($f_{iAw}(x)\downarrow$, resp.) and $\langle f_{iAw}(x), g_i(y)\rangle \sqsubseteq \langle f_{i+1Aw}(x), g_{i+1}(y)\rangle \sqsubseteq \ldots$ in S_w^A. Hence $g(y)\downarrow$ ($f_{Aw}(x)\downarrow$, resp.) and by admissibility of S_w^A also $\langle f_{Aw}(x), g(y)\rangle \in S_w^A$.

  Similar reasoning shows that $\langle f(x)_{Sw}, \mathsf{Rec}_{\mathsf{Loc}}(g)(\sigma)\rangle \in S_w^{St}$ whenever $\langle x_{Sw},\sigma\rangle \in R_w^{St}$ and either $f(x)\downarrow$ or $\mathsf{Rec}_{\mathsf{Loc}}(g)(\sigma)\downarrow$. Hence, $\langle f,g\rangle : R \subset S$ which we needed to show.

These properties are summarised in the following lemma.

Lemma 8.6.1. *$\mathcal{R}$ defines a normal relational structure on the bilimit-compact product category $\mathcal{C} \times$ **pCpo**. Inverse images f^*S are given by*

$$
\begin{array}{rcl}
(f^*S)_w^A & = & \{\langle x,y\rangle \in X_{Aw} \times Y \,\big|\, f'_{Aw}(x)\downarrow \vee f''(y)\downarrow \ \implies \ \langle f'_{Aw}(x), f''(y)\rangle \in S_{Aw}\} \\
(f^*S)_w^{St} & = & \left\{ \langle s,\sigma\rangle \in X_{Sw} \times \mathsf{Rec}_{\mathsf{Loc}}(Y) \,\middle|\, \begin{array}{l} f'_{Sw}(s)\downarrow \vee \mathsf{Rec}_{\mathsf{Loc}}(f'')(\sigma)\downarrow \implies \\ \qquad \langle f'_{Sw}(s), \mathsf{Rec}_{\mathsf{Loc}}(f'')(\sigma)\rangle \in R_w^{St} \end{array} \right\}
\end{array}
$$

Table 8.9 *The functional* Φ

At $A \in Type$, $w \in \mathcal{W}$ the map Φ is defined according to

$$\langle x, y\rangle \in \Phi(R,S)_w^{\mathsf{bool}} \overset{def}{\iff} y \in \mathsf{BVal} \text{ and } x = y$$

$$\langle r, s\rangle \in \Phi(R,S)_w^{\{\mathsf{m}_i : A_i\}} \overset{def}{\iff} s \in \mathsf{Rec}_{\mathcal{M}}(Y') \text{ and } \forall i \in I.\ s.\mathsf{m}_i\downarrow\ \wedge \langle r.\mathsf{m}_i, s.\mathsf{m}_i\rangle \in S_w^{A_i}$$

$$\begin{aligned}\langle f, g\rangle \in \Phi(R,S)_w^{A\Rightarrow B} \overset{def}{\iff}\ & g \in [\mathsf{Rec}_{\mathsf{Loc}}(Y) \times Y \to \mathsf{Rec}_{\mathsf{Loc}}(Y') \times Y'] \text{ and}\\ & \forall w' \geq w\ \forall \langle s, \sigma\rangle \in R_{w'}^{St}\ \forall \langle x, y\rangle \in R_{w'}^{A}\\ & \quad (f_{w'}(s,x)\uparrow\ \wedge g(\sigma, y)\uparrow) \text{ or}\\ & \quad (f_{w'}(s,x) = \langle w'', \langle s', x'\rangle\rangle \wedge g(\sigma, y) = \langle \sigma', y'\rangle\\ & \qquad \wedge \langle s', \sigma'\rangle \in S_{w''}^{St} \wedge \langle x', y'\rangle \in S_{w''}^{B})\end{aligned}$$

$$\langle x, y\rangle \in \Phi(R,S)_w^{\mathsf{ref}\ A} \overset{def}{\iff} y \in w \cap \mathsf{Loc}_A \text{ and } x = y$$

and at S_w it is given by

$$\langle s, \sigma\rangle \in \Phi(R,S)_w^{St} \overset{def}{\iff} \mathsf{dom}(s) = w = \mathsf{dom}(\sigma) \text{ and } \forall l_A \in w.\ \langle s.l, \sigma.l\rangle \in S_w^A$$

for all morphisms $f = \langle f', f''\rangle$ *with* $f' : X \to X'$, $f'' : Y \to Y'$ *and* $S \in \mathcal{R}(\langle X', Y'\rangle)$*; and* $\mathcal{R}$ *possesses intersections given componentwise by set-theoretic intersection. Moreover,* $\mathcal{R}_{adm} = \mathcal{R}$.

Next, in Table 8.9, we define a map $\Phi(R^-, R^+)$ on the relational structure corresponding to the equations for the Kripke logical relation R above (separating positive and negative occurrences) such that for $R \in \mathcal{R}(X,Y)$ and $S \in \mathcal{R}(X',Y')$ we have $\Phi(R,S) \in \mathcal{R}((F \times G)(\langle X, Y\rangle, \langle X', Y'\rangle))$.

It is not hard to show that Φ is well-defined, in the sense that it maps admissible relations to admissible relations: Admissibility of each S_w^{St} and S_w^A entails admissibility of the corresponding component of $\Phi(R,S)$:

Lemma 8.6.2. *Let* $R \in \mathcal{R}(X',Y')$ *and* $S \in \mathcal{R}(X,Y)$*. Then, for all* $w \in \mathcal{W}$ *and* $A \in Type$*,*

$$\Phi(R,S)_w^A \subseteq X_{Aw} \times Y \quad \textit{and} \quad \Phi(R,S)_w^{St} \subseteq X_{Sw} \times \mathsf{Rec}_{\mathsf{Loc}}(Y)$$

are admissible subsets of the cpos $X_{Aw} \times Y$ *and* $X_{Sw} \times \mathsf{Rec}_{\mathsf{Loc}}(Y)$*, respectively.*

Moreover, Φ defines an admissible action of the functor $F \times G$ on $\mathcal{R}$, in the following sense:

Lemma 8.6.3. *For all* $e = \langle e_1, e_2\rangle, f = \langle f_1, f_2\rangle$ *and* R, R', S, S'*, if* $e : R' \subset R$ *and* $f : S \subset S'$ *then* $(F \times G)(e,f) : \Phi(R,S) \subset \Phi(R',S')$.

According to Theorem 2.5.2, Lemma 8.6.3 guarantees that Φ has a unique fixed point $fix(\Phi)$ in $\mathcal{R}(D, \mathsf{Val})$, and we obtain the Kripke logical relation $R \overset{def}{=} fix(\Phi)$ satisfying $R = \Phi(R,R)$ as required.

Theorem 8.6.4 (Existence)**.** *The functional* Φ *has a unique fixed point.*

Proof of Lemma 8.6.3. Let $w \in \mathcal{W}$ and $A \in Type$. We consider cases for A.

· Case $A \equiv \mathsf{bool}$: By definition of F and G, $(F \times G)(e,f) = \langle F(e_1,f_1), G(e_2,f_2)\rangle$ with

$$F(e_1,f_1)_{\mathsf{bool}w} = \mathsf{id}_{\mathsf{BVal}} \quad \text{and} \quad G(e_2,f_2) = \mathsf{id}_{\mathsf{BVal}}$$

Now if $\langle x,y\rangle \in \Phi(R,S)^{\mathsf{bool}}_w$ then $y \in \mathsf{BVal}$ and $x = y$, hence,

$$\langle F(e_1,f_1)_{\mathsf{bool}w}(x), G(e_2,f_2)(y)\rangle = \langle x,y\rangle \in \Phi(R',S')^{\mathsf{bool}}_w$$

· Case $A \equiv \{\mathsf{m}_i : A_i\}$. Suppose $\langle x,y\rangle \in \Phi(R,S)^A_w$, i.e., $\langle x.\mathsf{m}_i, y.\mathsf{m}_i\rangle \in S^{A_i}_w$ for all i. By assumption, $f_{1\,A_i w}(x.\mathsf{m}_i)\downarrow$ if and only if $f_2(y.\mathsf{m}_i)$, and then

$$\langle f_{1\,A_i w}(x.\mathsf{m}_i), f_2(y.\mathsf{m}_i)\rangle \in {S'}^{A_i}_w \tag{8.6}$$

By definition of F and G, $F(e_1,f_1)_{Aw}(x)\downarrow$ if and only if $f_{1\,A_i w}(x.\mathsf{m}_i)\downarrow$ for all i, which by the above is equivalent to $f_2(y.\mathsf{m}_i)\downarrow$ for all i, i.e., $G(e_2,f_2)(y)\downarrow$, and then

$$F(e_1,f_1)_{Aw}(x).\mathsf{m}_i = f_{1\,A_i w}(x.\mathsf{m}_i) \quad \text{and} \quad G(e_2,f_2)(y).\mathsf{m}_i = f_2(y.\mathsf{m}_i)$$

Hence, (8.6) shows $\langle F(e_1,f_1)_{Aw}(x), G(e_2,f_2)(y)\rangle \in \Phi(R',S')^A_w$.

· $A \equiv B \Rightarrow B'$. Suppose $\langle h,k\rangle \in \Phi(R,S)^{B\Rightarrow B'}_w$, we have to show that

$$\langle F(e_1,f_1)_{B\Rightarrow B'\,w}(h), G(e_2,f_2)(k)\rangle \in \Phi(R',S')^{B\Rightarrow B'}_w \tag{8.7}$$

So let $w' \geq w$, $\langle s,\sigma\rangle \in {R'}^{St}_{w'}$ and $\langle x,y\rangle \in {R'}^{B}_{w'}$. By assumption,

$$\begin{array}{lll} e_{1\,Sw'}(s)\downarrow \iff \mathsf{Rec}_{\mathsf{Loc}}(e_2)(\sigma)\downarrow, & \text{and then} & \langle e_{1\,Sw'}(s), \mathsf{Rec}_{\mathsf{Loc}}(e_2)(\sigma)\rangle \in R^{St}_{w'} \\ e_{1\,Bw'}(x)\downarrow \iff e_2(y)\downarrow, & \text{and then} & \langle e_{1\,Bw'}(x), e_2(y)\rangle \in R^{B}_{w'} \end{array}$$

From $\langle h,k\rangle \in \Phi(R,S)^{B\Rightarrow B'}_w$ we then obtain

$$h_{w'}(e_{1\,Sw'}(s), e_{1\,Bw'}(x)) = \langle w'', \langle s',x'\rangle\rangle \iff k(\mathsf{Rec}_{\mathsf{Loc}}(e_2)(\sigma), e_2(y)) = \langle \sigma', y'\rangle$$

and then both $\langle s',\sigma'\rangle \in S^{St}_{w''}$ and $\langle x',y'\rangle \in S^{B'}_{w''}$. By assumption

$$\begin{array}{lll} f_{1\,Sw''}(s')\downarrow \iff \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma')\downarrow, & \text{and then} & \langle f_{1\,Sw''}(s'), \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma')\rangle \in {S'}^{St}_{w''} \\ f_{1\,B'w''}(x')\downarrow \iff f_2(y')\downarrow, & \text{and then} & \langle f_{1\,B'w''}(x'), f_2(y')\rangle \in {S'}^{B'}_{w''} \end{array}$$

By definition of F, G, we have

$$\begin{array}{rcl} F(f_1,e_1)(h)_{w'}(s,x)\downarrow & \Longrightarrow & \left\{\begin{array}{l} F(f_1,e_1)(h)_{w'}(s,x) \\ \quad = \langle w'', \langle f_{1\,Sw''}(s'), f_{1\,B'w''}(x')\rangle\rangle \end{array}\right. \\ G(e_2,f_2)(k)(\sigma,y)\downarrow & \Longrightarrow & G(e_2,f_2)(k)(\sigma,y) = \langle \mathsf{Rec}_{\mathsf{Loc}}(f_2)(\sigma'), f_2(y')\rangle \end{array}$$

so by the above considerations (8.7) holds.

· Case $A \equiv \mathsf{ref}\ B$. By definition of F and G,

$$F(e_1,f_1)_{\mathsf{ref}\ B\,w} = \mathsf{id}_{w\cap \mathsf{Loc}_B} \quad \text{and} \quad G(e_2,f_2) = \mathsf{id}_{\mathsf{Loc}}$$

So if $\langle x,y\rangle \in \Phi(R,S)^{\mathsf{ref}\ B}_w$ then necessarily $y \in w \cap \mathsf{Loc}_A$ and $x = y$, hence,

$$\langle F(e_1,f_1)_{\mathsf{ref}\ B\,w}(x), G(e_2,f_2)(y)\rangle = \langle x,y\rangle \in \Phi(R',S')^{\mathsf{ref}\ B}_w$$

Finally, the case for $\langle s,\sigma\rangle \in \Phi(R,S)^{St}_w$ proceeds analoguously to the case for record types $\{\mathsf{m}_i : A_i\}$ above. □

8.6.2 The Basic Lemma

We establish the following two monotonicity properties before proving the basic lemma of logical relations for the Kripke logical relation R:

Lemma 8.6.5 (Kripke Monotonicity)**.** *Let $w, w' \in \mathcal{W}$ such that $w' \geq w$, and let $\langle a, u\rangle \in R^A_w$. Then $\langle [\![A]\!]^{w'}_w\,(a), u\rangle \in R^A_{w'}$.*

Proof. By induction on A (note that this is possible here because, in the case of function types $A \Rightarrow B$, the map $[\![A \Rightarrow B]\!]^{w'}_w$ does *not* depend on the store).

- Case $A \equiv \mathsf{bool}$. This follows immediately from $[\![\mathsf{bool}]\!]^{w'}_w\,(x) = \mathsf{id}(x) = x$.
- Case $A \equiv \{\mathsf{m}_i : A_i\}_{i\in I}$. By definition of R^A_w we know $y \in \mathsf{Rec}_{\mathcal{M}}(\mathsf{Val})$ and $\langle x.\mathsf{m}_i, y.\mathsf{m}_i\rangle \in R^{A_i}_w$ for all $i \in I$. So by induction hypothesis, $\langle [\![A_i]\!]^{w'}_w\,(x.\mathsf{m}_i), y.\mathsf{m}_i\rangle \in R^{A_i}_w$ for all i, and $\langle [\![A]\!]^{w'}_w\,(x), y\rangle \in R^A_{w'}$ follows since

$$[\![A]\!]^{w'}_w\,(x).\mathsf{m}_i = [\![A_i]\!]^{w'}_w\,(x.\mathsf{m}_i)$$

- Case $A \equiv B \Rightarrow B'$. By definition, $[\![B \Rightarrow B']\!]^{w'}_w\,(x) = \lambda_{w''\geq w'} x_{w''}$, so the result follows directly from the definition of $R^{B\Rightarrow B'}_{w'}$ and the assumption $\langle x, y\rangle \in R^{B\Rightarrow B'}_w$.
- Case $A \equiv \mathsf{ref}\ B$. Immediately from $[\![\mathsf{ref}\ B]\!]^{w'}_w\,(x) = x$.

□

Lemma 8.6.6 (Subtype Monotonicity)**.** *Let $w \in \mathcal{W}$, let $A \preceq B$ and suppose $\langle a, u\rangle \in R^A_w$. Then $\langle [\![A \preceq B]\!]_w\,(a), u\rangle \in R^B_w$.*

Proof. By a straightforward induction on the derivation of $A \preceq B$: Suppose $\langle x, y\rangle \in R^A_w$. If the last step in the derivation $A \preceq B$ is by

- (Reflexivity). In this case, $A \equiv B$ and $[\![A \preceq B]\!]_w\,(x) = x$, so that $\langle [\![A \preceq B]\!]_w\,(x), y\rangle \in R^B_w$ is immediate.
- (Transitivity). Assume $A \preceq B$ was derived from $A \preceq A'$ and $A' \preceq B$. Applying the induction hypothesis, $\langle [\![A \preceq A']\!]_w\,(x), y\rangle \in R^{A'}_w$ and again by induction hypothesis,

$$\langle [\![A' \preceq B]\!]_w\,([\![A \preceq A']\!]_w\,(x)), y\rangle \in R^B_w$$

  as required.

- (Arrow). Write $x' \stackrel{def}{=} [\![A\Rightarrow B \preceq A'\Rightarrow B']\!]_w\,(x)$, we must show $\langle x', y\rangle \in R^{A'\Rightarrow B'}_w$. Let $w' \geq w$, $\langle s, \sigma\rangle \in R^{St}_{w'}$ and $\langle u, u'\rangle \in R^{A'}_{w'}$.

  By induction, $\langle [\![A' \preceq A]\!]_{w'}\,(u), u'\rangle \in R^A_{w'}$, and therefore we have

$$x_{w'}(s, [\![A' \preceq A]\!]_{w'}\,(v))\downarrow \quad\Longleftrightarrow\quad y(\sigma, v')\downarrow$$

  by assumption $\langle x, y\rangle \in R^{A\Rightarrow B}_w$. Moreover, if both are defined then

$$x_{w'}(s, [\![A' \preceq A]\!]_{w'}\,(u)) = \langle w'', \langle s', v\rangle\rangle \quad \text{and} \quad y(\sigma, u') = \langle \sigma', v'\rangle$$

  such that $\langle s', \sigma'\rangle \in R^{St}_{w''}$ and $\langle v, v'\rangle \in R^B_{w''}$. By induction hypothesis, the latter entails $\langle [\![B \preceq B']\!]_{w''}\,(v), v'\rangle \in R^{B'}_{w''}$ and we can conclude $\langle x', y\rangle \in R^{A'\Rightarrow B'}_w$.

- (Record). Suppose $\{\mathsf{m}_i : A_i\}_{i\in I} \preceq \{\mathsf{m}_i : A'_i\}_{i\in I'}$ has been derived from hypotheses $I' \subseteq I$ and $A_i \preceq A'_i$, for all $i \in I'$. Assume $\langle x, y\rangle \in R_w^{\{\mathsf{m}_i:A_i\}_{i\in I}}$. If we let $x' \stackrel{def}{=} [\![\{\mathsf{m}_i : A_i\}_{i\in I} \preceq \{\mathsf{m}_i : A'_i\}_{i\in I'}]\!]_w\,(x)$ we must show $\langle x', y\rangle \in R_w^{\{\mathsf{m}_i:A'_i\}_{i\in I'}}$.

  By assumption $y \in \mathsf{Rec}_{\mathcal{M}}(\mathsf{Val})$ and $\langle x.\mathsf{m}_i, y.\mathsf{m}_i\rangle \in R_w^{A_i}$, for all $i \in I$. By induction hypothesis, $\langle [\![A_i \preceq A'_i]\!]_w\,(x.\mathsf{m}_i), y.\mathsf{m}_i\rangle \in R_w^{A_i}$ for all $i \in I' \subseteq I$. The result follows, since by definition $x'.\mathsf{m}_i = [\![A_i \preceq A'_i]\!]_w\,(x.\mathsf{m}_i)$.

□

Lemma 8.6.5 and Lemma 8.6.6 show a key property of the relation R, which lies at the heart of the coherence proof: For $\langle a, u\rangle \in R_w^A$ we can apply coercions to a and enlarge the world w while remaining in relation with $u \in \mathsf{Val}$.

We extend R to contexts Γ in the natural way, by defining

$$\langle \rho, \eta\rangle \in R_w^\Gamma \quad \stackrel{def}{\iff} \quad \langle \rho(x), \eta(x)\rangle \in R_w^A \quad \forall x{:}A \text{ in } \Gamma$$

It is tedious, but not difficult, to prove the fundamental property of logical relations. It states that the (typed and untyped) denotations of well-typed terms compute related results.

Lemma 8.6.7 (Basic Lemma)**.** *Suppose* $\Gamma \rhd e : A$, $w \in \mathcal{W}$, $\langle \rho, \eta\rangle \in R_w^\Gamma$ *and* $\langle s, \sigma\rangle \in R_w^{St}$. *Then*

- *either* $[\![\Gamma \rhd e : A]\!]_w\,\rho s \uparrow$ *and* $[\![e]\!]\,\eta\sigma \uparrow$, *or*
- *there are* $w' \geq w, s', a, \sigma', u$ *such that* $[\![\Gamma \rhd e : A]\!]_w\,\rho s = \langle w', \langle s', a\rangle\rangle$ *and* $[\![e]\!]\,\eta\sigma = \langle \sigma', u\rangle$ *such that* $\langle s', \sigma'\rangle \in R_{w'}^{St}$ *and* $\langle a, u\rangle \in R_{w'}^A$.

Proof. The proof is by induction on the derivation of $\Gamma \rhd e : A$, using Lemmas 8.6.5 and 8.6.6.

- (Subsumption). In this case, $\Gamma \rhd e : B$ has been derived from antecedent $\Gamma \rhd e : A$ and $A \preceq B$. By induction hypothesis, either both $[\![\Gamma \rhd e : A]\!]_w\,\rho s \uparrow$ and $[\![e]\!]\,\eta\sigma \uparrow$, or $[\![\Gamma \rhd e : A]\!]_w\,\rho s = \langle w', \langle s', x\rangle\rangle$ and $[\![e]\!]\,\eta\sigma = \langle \sigma', y\rangle$ where $\langle s', \sigma'\rangle \in R_{w'}^{St}$ and $\langle x, y\rangle \in R_{w'}^A$. But then $\langle [\![A \preceq B]\!]_{w'}\,(x), y\rangle \in R_{w'}^B$, by the previous Lemma, which concludes this case using the semantics of the subsumption rule.
- (Var). By assumption $\langle \rho, \eta\rangle \in R_w^\Gamma$, and by the premiss of the variable rule $x : A \in \Gamma$, which entails $\langle \rho(x), \eta(x)\rangle \in R_w^A$. The result now follows immediately from the definitions of $[\![\Gamma \rhd x : A]\!]$ and $[\![x]\!]$, and the assumption $\langle s, \sigma\rangle \in R_w^{St}$.
- (Let). Assume we have derived $\Gamma \rhd \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : B$ by the rule (Let). By induction hypothesis, either both $[\![\Gamma \rhd e_1 : A]\!]_w\,\rho s \uparrow$ and $[\![e_1]\!]\,\eta\sigma \uparrow$, or $[\![\Gamma \rhd e_1 : A]\!]_w\,\rho s = \langle w', \langle s', u\rangle\rangle$ and $[\![e_1]\!]\,\eta\sigma = \langle \sigma', v\rangle$ where $\langle s', \sigma'\rangle \in R_{w'}^{St}$ and $\langle u, v\rangle \in R_{w'}^A$.

  In the latter case, observe that $\langle [\![\Gamma]\!]_w^{w'}\,(\rho), \eta\rangle \in R_{w'}^\Gamma$ by Kripke Monotonicity, and therefore $\langle [\![\Gamma]\!]_w^{w'}\,(\rho)[x := u], \eta[x := v]\rangle \in R_{w'}^{\Gamma,x:A}$. Applying the inductive hypothesis to the derivation $\Gamma, x : A \rhd e_2 : B$ we obtain that either both

  $$[\![e_2]\!]\,\eta[x := v]\sigma' \uparrow \quad \text{and} \quad [\![\Gamma, x{:}A \rhd e_2 : B]\!]_{w'}\,([\![\Gamma]\!]_w^{w'}\,(\rho)[x := u])s' \uparrow$$

  are undefined, or both are defined:

- $[\![\Gamma, x{:}A \rhd e_2 : B]\!]_{w'}\,([\![\Gamma]\!]_w^{w'}\,(\rho)[x := u])s' = \langle w'', \langle s'', u'\rangle\rangle$
- $[\![e_2]\!]\,\eta[x := v]\sigma' = \langle \sigma'', v'\rangle$

where $\langle s'', \sigma''\rangle \in R^{St}_{w''}$ and $\langle u', v'\rangle \in R^{B}_{w''}$. Using the definition of $[\![\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2]\!]$ and $[\![\Gamma \rhd \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : B]\!]$, this is all that needed to be shown.

· (Const) Suppose we have derived $\Gamma \rhd \mathsf{true} : \mathsf{bool}$ by the rule for constant true. The result follows directly from $[\![\Gamma \rhd \mathsf{true} : \mathsf{bool}]\!]_w\,\rho s = \langle s, \mathit{true}\rangle$ and $[\![\mathsf{true}]\!]\,\eta\sigma = \langle \sigma, \mathit{true}\rangle$, the assumption $\langle s, \sigma\rangle \in R^{St}_w$ and the definition of R^{bool}_w. The case where $\Gamma \rhd \mathsf{false} : \mathsf{bool}$ is analogous.

· (If) By induction hypothesis on the premiss $\Gamma \rhd x : \mathsf{bool}$, the assumption $\langle \rho, \eta\rangle \in R^{\Gamma}_w$ and the definition of the semantics, $[\![\Gamma \rhd x : \mathsf{bool}]\!]_w\,\rho s = \langle w, \langle s, u\rangle\rangle$ and $[\![x]\!]\,\eta\sigma = \langle \sigma, v\rangle$ s.t. $\langle u, v\rangle \in R^{\mathsf{bool}}_w$, for all $\langle s, \sigma\rangle \in R^{St}_w$. By definition this means $u, v \in \mathsf{BVal}$ and $u = v$.

We consider the case where $u = \mathit{true} = v$, the case where both equal *false* is analogous. By induction hypothesis on $\Gamma \rhd e_1 : A$, either both $[\![\Gamma \rhd e_1 : A]\!]_w\,\rho s \uparrow$ and $[\![e_1]\!]\,\eta\sigma \uparrow$, or $[\![\Gamma \rhd e_1 : A]\!]_w\,\rho s = \langle w', \langle s', u'\rangle\rangle$ and $[\![e_1]\!]\,\eta\sigma = \langle \sigma', v'\rangle$ where $\langle s', \sigma'\rangle \in R^{St}_{w'}$ and $\langle u', v'\rangle \in R^{A}_{w'}$. The result follows by observing that

$$[\![\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A]\!]_w\,\rho s = \langle w', \langle s', u'\rangle\rangle$$

and $[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\,\eta\sigma = \langle \sigma', v'\rangle$.

· (Record) For all $i \in I$, by induction hypothesis and from the fact that $[\![x_i]\!]\,\eta\sigma = \langle \sigma, \eta(x_i)\rangle$ one obtains $[\![\Gamma \rhd x_i : A_i]\!]_w\,\rho s = \langle w, \langle s, u_i\rangle\rangle$ s.t. $\langle u_i, \eta(x_i)\rangle \in R^{A_i}_w$.

From the semantic equations, $[\![\Gamma \rhd \{\mathsf{m}_i = x_i\} : \{\mathsf{m}_i : A_i\}]\!]_w\,\rho s = \langle w, \langle s, \{\!|\mathsf{m}_i = u_i|\!\}\rangle\rangle$ and also $[\![\{\mathsf{m}_i = x_i\}]\!]\,\eta\sigma = \langle \sigma, \{\!|\mathsf{m}_i = \eta(x_i)|\!\}\rangle$, so the result follows directly from the definition of $R^{\{\mathsf{m}_i:A_i\}}_w$.

· (Selection) By induction hypothesis we have $\langle u, \eta(x)\rangle \in R^{\{\mathsf{m}_i:A_i\}}_w$ under the assumption that $\langle w, \langle s, u\rangle\rangle = [\![\Gamma \rhd x : \{\mathsf{m}_i : A_i\}]\!]_w\,\rho s$. Therefore, the definition of $R^{\{\mathsf{m}_i:A_i\}}_w$ entails $\langle u.\mathsf{m}_j, \eta(x).\mathsf{m}_j\rangle \in R^{A_j}_w$ and the result follows from the semantics of $x.\mathsf{m}_j$.

· (Lambda). This case is similar to the case for (Let), using Kripke Monotonicity when constructing an extended context.

We know that there exist f, g s.t. $[\![\Gamma \rhd \lambda x.e : A \Rightarrow B]\!]_w\,\rho s = \langle s, f\rangle$ and $[\![\lambda x.e]\!]\,\eta\sigma = \langle \sigma, g\rangle$, so all that needs to be shown is $\langle f, g\rangle \in R^{A\Rightarrow B}_w$. Let $w' \geq w$, $\langle u, v\rangle \in R^{A}_{w'}$ and $\langle s', \sigma'\rangle \in R^{St}_{w'}$. By Lemma 8.6.5, $\langle [\![\Gamma]\!]_w^{w'}\,(\rho)[x := u], \eta[x := v]\rangle \in R^{\Gamma, x:A}_{w'}$. By induction hypothesis for the premiss $\Gamma, x{:}A \rhd e : B$, either both

$$[\![\Gamma, x{:}A \rhd e : B]\!]_{w'}\,([\![\Gamma]\!]_w^{w'}\,(\rho)[x := u])s' \uparrow \quad \text{and} \quad [\![e]\!]\,\eta[x := v]\sigma' \uparrow$$

or else there are w'', s'', u', σ'' and v' for which

$$[\![\Gamma, x{:}A \rhd e : B]\!]_{w'}\,([\![\Gamma]\!]_w^{w'}\,(\rho)[x := u])s' = \langle w'', \langle s'', u'\rangle\rangle$$

and $[\![e]\!]\,\eta[x := v]\sigma' = \langle \sigma'', v'\rangle$ where $\langle s'', \sigma''\rangle \in R^{St}_{w''}$ and $\langle u', v'\rangle \in R^{B}_{w''}$. But this is just the definition of $\langle f, g\rangle \in R^{A\Rightarrow B}_w$.

· (Application) Suppose the derivation of $\Gamma \rhd x(y) : B$ ends with the application rule. By the premiss of the rule, $\Gamma \rhd x : A \Rightarrow B$ and $\Gamma \rhd y : A$, so by induction $[\![\Gamma \rhd x : A \Rightarrow B]\!]_w\,\rho s = \langle w, \langle s, f\rangle\rangle$ s.t. $\langle f, \eta(x)\rangle \in R^{A\Rightarrow B}_w$, and $[\![\Gamma \rhd y : A]\!]_w\,\rho s = \langle w, \langle s, u\rangle\rangle$ s.t. $\langle u, \eta(y)\rangle \in R^{A}_w$.

By definition of $R_w^{A\Rightarrow B}$, either both $[\![\Gamma \triangleright x(y) : B]\!]_w\, \rho s = f_w(s,u)\uparrow$ and $[\![x(y)]\!]\, \eta\sigma = \eta(x)(\sigma, \eta(y))\uparrow$, or

$$[\![\Gamma \triangleright x(y) : B]\!]_w\, \rho s = \langle w', \langle s', u'\rangle\rangle \text{ and } [\![x(y)]\!]\, \eta\sigma = \langle \sigma', v'\rangle$$

with $\langle s', \sigma'\rangle \in R_{w'}^{St}$ and $\langle u', v'\rangle \in R_{w'}^{B}$.

- (New) By definition, we have

$$[\![\Gamma \triangleright \mathsf{new}_A\ x : \mathsf{ref}\ A]\!]_w\, \rho s = \langle w, l_A, \langle s', l_A\rangle\rangle \quad \text{and} \quad [\![\mathsf{new}_A\ x]\!] = \langle \sigma', l_A\rangle$$

with $l_A \in \mathsf{Loc}_A$ and s', σ' as in Table 8.6 in Section 8.4 and the semantic equations given in Table 8.7 in Section 8.5, resp. All that remains to show is $\langle s', \sigma'\rangle \in R_{w'}^{St}$, where $w' = w, l_A$.

From the assumption $\langle s, \sigma\rangle \in R_w^{St}$ we immediately find $\mathsf{dom}(s') = w' = \mathsf{dom}(\sigma')$. Moreover, by induction we have $[\![\Gamma \triangleright x : A]\!]_w\, \rho s = \langle w, \langle s, u\rangle\rangle$ with $\langle u, \eta(x)\rangle \in R_w^A$. By Kripke Monotonicity this entails $\langle [\![A]\!]_w^{w'}(u), \eta(x)\rangle \in R_{w'}^A$. Also, by assumption $\langle s, \sigma\rangle \in R_w^{St}$ we have $\langle s.l, \sigma.l\rangle \in R_w^{A'}$ for all $l \in w \cap \mathsf{Loc}_{A'}$. By Kripke Monotonicity this gives $\langle [\![A']\!]_w^{w'}(s.l), \sigma.l\rangle \in R_{w'}^{A'}$, and we have shown $\langle s'.l', \sigma'.l'\rangle \in R_{w'}^{A'}$ for all $l' \in w'$. Thus $\langle s', \sigma'\rangle \in R_{w'}^{St}$ as required.

- (Deref) By induction hypothesis, $[\![\Gamma \triangleright x : \mathsf{ref}\ A]\!]_w\, \rho s = \langle w, \langle s, l\rangle\rangle$ s.t. $\langle l, \eta(x)\rangle \in R_w^{\mathsf{ref}\ A}$, i.e., $\eta(x) \in w \cap \mathsf{Loc}_A$ and $l = \eta(x)$. Then $\langle s, \sigma\rangle \in R_w^{St}$ shows $\langle s.l, \sigma.l\rangle \in R_w^A$. The result follows since $[\![\Gamma \triangleright \mathsf{deref}\ x : A]\!]_w\, \rho s = \langle w, \langle s, s.l\rangle\rangle$ and $[\![\mathsf{deref}\ x]\!]\, \eta\sigma = \langle \sigma, \sigma.\eta(x)\rangle$.

- (Update) By definition and the assumptions $\langle \rho, \eta\rangle \in R_w^\Gamma$ and $\langle s, \sigma\rangle \in R_w^{St}$ we necessarily have $[\![x{:=}y]\!]\, \eta\sigma = \langle \sigma', \{\!|\,|\!\}\rangle$ and $[\![\Gamma \triangleright x{:=}y : \mathbf{1}]\!]_w\, \rho s = \langle w, \langle s', \{\!|\,|\!\}\rangle\rangle$, with s' and σ' as in Tables 8.6 and 8.7, resp.

Clearly $\langle \{\!|\,|\!\}, \{\!|\,|\!\}\rangle \in R_w^{\mathbf{1}}$, and all that remains to be shown is $\langle s', \sigma'\rangle \in R_w^{\mathbf{1}}$. From $\langle s, \sigma\rangle \in R_w^{St}$ and the definition of s', σ' one sees $\mathsf{dom}(s') = w = \mathsf{dom}(\sigma')$. So let $l \in w \cap \mathsf{Loc}_B$. Then, by the induction hypothesis, $[\![\Gamma \triangleright x : \mathsf{ref}\ A]\!]_w\, \rho s = \langle w, \langle s, l_0\rangle\rangle$ s.t. $\langle l_0, \eta(x)\rangle \in R_w^{\mathsf{ref}\ A}$, i.e., $l_0 = \eta(x) \in w \cap \mathsf{Loc}_A$.

Thus, if $l \neq l_0$ then $\langle s'.l, \sigma'.l\rangle = \langle s.l, \sigma.l\rangle \in R_w^B$, by assumption $\langle s, \sigma\rangle \in R_w^{St}$. Finally, for $l = l_0$ we have $\langle s'.l, \sigma'.l\rangle = \langle u, \eta(y)\rangle \in R_w^A$, since by induction hypothesis, $[\![\Gamma \triangleright y : A]\!]_w\, \rho s = \langle w, \langle s, u\rangle\rangle$ with $\langle u, \eta(y)\rangle \in R_w^A$.

□

8.6.3 Bracketing

Next, in Table 8.10, we define families of "bracketing" maps ϕ_w, ψ_w,

$$[\![A]\!]_w \underset{\psi_w^A}{\overset{\phi_w^A}{\rightleftarrows}} \mathsf{Val} \quad \text{and} \quad S_w \underset{\psi_w^{St}}{\overset{\phi_w^{St}}{\rightleftarrows}} \mathsf{St}$$

such that $\psi_w^A \circ \phi_w^A = \mathsf{id}_{[\![A]\!]_w}$, i.e., each $[\![A]\!]_w$ is a retract of the untyped model Val. As in (Reynolds 2002b), the retraction property follows from a more general result which justifies the term "bracketing",

$$\phi_w^A \subseteq R_w^A \quad \text{and} \quad R_w^A \subseteq (\psi_w^A)^{op}$$

relating the (graphs of the) bracketing maps and the Kripke logical relation of the previous section.

Table 8.10 *Bracketing maps*

$$
\begin{array}{lcl}
\phi_w^{\mathsf{bool}}(b) & = & b \\
\psi_w^{\mathsf{bool}}(v) & = & \begin{cases} v & \text{if } v \in \mathsf{BVal} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\phi_w^{\{\mathsf{m}_i:A_i\}}(r) & = & \{\!|\mathsf{m}_i = \phi_w^{A_i}(r.\mathsf{m}_i)|\!\} \\
\psi_w^{\{\mathsf{m}_i:A_i\}}(v) & = & \begin{cases} \{\!|\mathsf{m}_i = \psi_w^{A_i}(v.\mathsf{m}_i)|\!\} & \\ \qquad\qquad \text{if } v \in \mathsf{Rec}_{\mathbb{L}}(\mathsf{Val}) \text{ and } \psi_w^{A_i}(v.\mathsf{m}_i)\downarrow \text{ for all } i \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
\phi_w^{A\Rightarrow B}(f) & = & \lambda\langle\sigma,v\rangle.\begin{cases} \langle \phi_{w''}^{St}(s), \phi_{w''}^{B}(b)\rangle & \\ \qquad\qquad \text{if } \mathsf{dom}(\sigma) = w' \in \mathcal{W}, \psi_{w'}^{St}(\sigma)\downarrow, \psi_{w'}^{A}(v)\downarrow & \\ \qquad\qquad \text{and } f_{w'}(\psi_{w'}^{St}(\sigma), \psi_{w'}^{A}(v)) = \langle w'', \langle s, b\rangle\rangle & \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
\psi_w^{A\Rightarrow B}(g) & = & \lambda_{w'\geq w}\,\lambda\langle s,a\rangle.\begin{cases} \langle w'', \langle \psi_{w''}^{St}(\sigma), \psi_{w''}^{B}(v)\rangle\rangle & \\ \qquad\qquad \text{if } g(\phi_{w'}^{St}(s), \phi_{w'}^{A}(a)) = \langle\sigma,v\rangle & \\ \qquad\qquad \mathsf{dom}(\sigma) = w'' \in \mathcal{W}, & \\ \qquad\qquad \psi_{w''}^{St}(\sigma)\downarrow \text{ and } \psi_{w''}^{B}(v)\downarrow & \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
\phi_w^{\mathsf{ref}\,A}(l) & = & l \\
\psi_w^{\mathsf{ref}\,A}(v) & = & \begin{cases} v & \text{if } v \in \mathsf{Loc}_A \\ \text{undefined} & \text{otherwise} \end{cases} \\
\phi_w^{St}(s) & = & \{\!|l_A = \phi_w^A(s.l_A)|\!\}_{l_A\in w} \\
\psi_w^{St}(\sigma) & = & \begin{cases} \{\!|l_A = \psi_w^A(\sigma.l_A)|\!\}_{l_A\in w} & \text{if } \psi_w^A(\sigma.l_A)\downarrow \text{ for all } l_A \in w \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Theorem 8.6.8 (Bracketing)**.** *For all $w \in \mathcal{W}$ and $A \in Type$,*

1. *for all $x \in [\![A]\!]_w$. $\langle x, \phi_w^A(x)\rangle \in R_w^A$,*
2. *for all $s \in S_w$. $\langle s, \phi_w^{St}(s)\rangle \in R_w^{St}$*
3. *for all $\langle x, y\rangle \in R_w^A$. $x = \psi_w^A(y)$,*
4. *for all $\langle s, \sigma\rangle \in R_w^{St}$. $s = \psi_w^{St}(\sigma)$*

Compared to Reynolds work, the proof of Theorem 8.6.8 is more involved, again due to the (mixed-variant) type recursion caused by the use of higher-order store. Therefore we first show a preliminary lemma, which uses the projection maps that come with the minimal invariant solution D of the endofunctor F on C: For $\delta(e) = F(e,e)$ we set $\pi_n^{Aw} \stackrel{def}{=} \delta^n(\bot)_{Aw}$, and similarly $\pi_n^{Sw} \stackrel{def}{=} \delta^n(\bot)_{Sw}$. Note that by definition of the minimal invariant solution,

$$\bigsqcup_n \pi_n^{Aw} = (\bigsqcup_n \delta^n(\bot))_{Aw} = (\mathsf{lfp}(\delta))_{Aw} = \mathsf{id}_{Aw}$$

follows. Similarly, $\bigsqcup_n \pi_n^{Sw} = \mathsf{id}_{Sw}$ holds.

Lemma 8.6.9. *For all $n \in \mathbb{N}$, $w \in \mathcal{W}$, $A \in Type$,*

1. $\forall x \in [\![A]\!]_w \,.\, \pi_n^{Aw}(x)\downarrow \implies \langle \pi_n^{Aw}(x), \phi_w^A(\pi_n^{Aw}(x))\rangle \in R_w^A$
2. $\forall s \in S_w.\ \pi_n^{Sw}(s)\downarrow \implies \langle \pi_n^{Sw}(s), \phi_w^{St}(\pi_n^{Sw}(s))\rangle \in R_w^{St}$
3. $\forall \langle x, y\rangle \in R_w^A.\ \pi_n^{Aw}(x)\downarrow \implies \pi_n^{Aw}(x) = \pi_n^{Aw}(\psi_w^A(y))$
4. $\forall \langle s, \sigma\rangle \in R_w^{St}.\ \pi_n^{Sw}(s)\downarrow \implies \pi_n^{Sw}(s) = \pi_n^{Sw}(\psi_w^{St}(\sigma))$

Proof. By a simultaneous induction on n, considering cases for A in parts 1 and 3. Clearly the result holds for $n = 0$ since then π_0^{Aw} and π_0^{Sw} are undefined everywhere. For the case $n > 0$:

1. We consider cases for A:

   · Case $A \equiv \mathsf{bool}$: By definition, $\pi_n^{\mathsf{bool}w}(x) = x \in \mathsf{BVal}$. Therefore $\phi_w^{\mathsf{bool}}(\pi_n^{\mathsf{bool}w}(x)) = \pi_n^{\mathsf{bool}w}(x) = x \in \mathsf{BVal}$. Hence,

   $$\langle \pi_n^{\mathsf{bool}w}(x), \phi_w^{\mathsf{bool}}(\pi_n^{\mathsf{bool}w}(x))\rangle = \langle x, x\rangle \in R_w^{\mathsf{bool}}$$

   by the definition of R_w^{bool}.

   · Case $A \equiv \{\!|\mathsf{m}_i : A_i|\!\}$: We know $\pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}}(x) = \{\!|\mathsf{m}_i = \pi_{n-1}^{A_i w}(x.\mathsf{m}_i)|\!\}$. By induction hypothesis,

   $$\langle \pi_{n-1}^{A_i w}(x.\mathsf{m}_i), \phi_w^{A_i}(\pi_{n-1}^{A_i w}(x.\mathsf{m}_i))\rangle \in R_w^{A_i}$$

   for all i and, by the definition of $\pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}w}$ and $\phi_w^{\{\!|\mathsf{m}_i:A_i|\!\}}$,

   $$\begin{aligned}\phi_w^{\{\!|\mathsf{m}_i:A_i|\!\}}(\pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}}(x)) &= \phi_w^{\{\!|\mathsf{m}_i:A_i|\!\}}(\{\!|\mathsf{m}_i = \pi_{n-1}^{A_i w}(x.\mathsf{m}_i)|\!\}) \\ &= \{\!|\mathsf{m}_i = \phi_w^{A_i}(\pi_{n-1}^{A_i w}(x.\mathsf{m}_i))|\!\}\end{aligned}$$

   Therefore $\langle \pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}w}(x).\mathsf{m}_i, \phi_w^{\{\!|\mathsf{m}_i:A_i|\!\}}(\pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}w}(x)).\mathsf{m}_i\rangle \in R_w^{A_i}$ for all i, i.e.,

   $$\langle \pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}w}(x), \phi_w^{\{\!|\mathsf{m}_i:A_i|\!\}}(\pi_n^{\{\!|\mathsf{m}_i:A_i|\!\}w}(x))\rangle \in R_w^{\{\!|\mathsf{m}_i:A_i|\!\}}$$

   as required.

   · Case $A \equiv B \Rightarrow B'$: By definition of $\pi_n^{B\Rightarrow B'\,w}$ and $\phi_w^{B\Rightarrow B'}$,

   $$\pi_n^{B\Rightarrow B'\,w}(x) = \lambda w' \geq w\, \lambda\langle s, v\rangle. \begin{cases} \langle \pi_{n-1}^{Sw''}(s'), \pi_{n-1}^{B'w''}(v')\rangle & \\ \qquad \text{if } x_{w'}(\pi_{n-1}^{Sw'}(s), \pi_{n-1}^{Bw'}(v)) & \\ \qquad\quad = \langle w'', \langle s', v'\rangle\rangle & \\ \qquad \text{and } \pi_{n-1}^{Sw''}(s')\downarrow & \\ \qquad \text{and } \pi_{n-1}^{B'w''}(v')\downarrow & \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

   and $\phi_w^{B\Rightarrow B'}(\pi_n^{B\Rightarrow B'\,w}(x))$ equals

   $$\lambda\langle \sigma, v\rangle. \begin{cases} \langle \phi_{w''}^{St}(\pi_{n-1}^{Sw''}(s')), \phi_{w''}^{B'}(\pi_{n-1}^{B'w''}(v'))\rangle & \\ \qquad \text{if } \mathsf{dom}(\sigma) = w' \geq w, \psi_{w'}^{St}(\sigma)\downarrow, \psi_{w'}^{B}(v)\downarrow & \\ \qquad \text{and } x_{w'}(\pi_{n-1}^{Sw'}(\psi_{w'}^{St}(s)), \pi_{n-1}^{Bw'}(\psi_{w'}^{B}(v))) = \langle w'', \langle s', v'\rangle\rangle & \\ \text{undefined} & \\ \qquad \text{otherwise} \end{cases}$$

   In order to show $\langle \pi_n^{B\Rightarrow B'\,w}(x), \phi_w^{B\Rightarrow B'}(\pi_n^{B\Rightarrow B'\,w}(x))\rangle \in R_w^{B\Rightarrow B'}$, let $w' \geq w$, $\langle s, \sigma\rangle \in R_{w'}^{St}$ and $\langle u, v\rangle \in R_{w'}^{B}$.

If either of $\pi_{n-1}^{Sw'}(s)$ or $\pi_{n-1}^{Bw'}(u)$ is undefined then both $\pi_n^{B\Rightarrow B'\,w}(x)_{w'}(s,u)\uparrow$ and $\phi_w^{B\Rightarrow B'}(\pi_n^{B\Rightarrow B'\,w}(x))_{w'}(\sigma,\nu)\uparrow$. So without loss of generality assume both $\pi_{n-1}^{Sw'}(s)\downarrow$ and $\pi_{n-1}^{Bw'}(u)\downarrow$ in the following. Then, by part 3 and 4, resp., of the induction hypothesis, $\pi_{n-1}^{Bw'}(u) = \pi_{n-1}^{Bw'}(\psi_{w'}^{B}(\nu))$ and $\pi_{n-1}^{Sw'}(s) = \pi_{n-1}^{Sw'}(\psi_{w'}^{St}(\sigma))$. Now if

$$x_{w'}(\pi_{n-1}^{Sw'}(s), \pi_{n-1}^{Bw'}(u)) = \langle w'', \langle s', u'\rangle\rangle$$

then the induction hypothesis entails firstly $\langle \pi_{n-1}^{Sw''}(s'), \phi_{w''}^{St}(\pi_{n-1}^{Sw''}(s'))\rangle \in R_{w''}^{St}$, and secondly $\langle \pi_{n-1}^{B'w''}(\nu'), \phi_{w''}^{B'}(\pi_{n-1}^{B'w''}(\nu'))\rangle \in R_{w''}^{B'}$ whenever $\pi_{n-1}^{Sw''}(s')\downarrow$ and $\pi_{n-1}^{B'w''}(\nu')\downarrow$ both hold.
Thus we have established $\langle \pi_n^{B\Rightarrow B'\,w}(x), \phi_w^{B\Rightarrow B'}(\pi_n^{B\Rightarrow B'\,w}(x))\rangle \in R_w^{B\Rightarrow B'}$, by definition of $R_w^{B\Rightarrow B'}$.

- Case $A \equiv \mathsf{ref}\ B$: By definition, $\pi_n^{\mathsf{ref}\ Bw}(x) = x \in \mathsf{Loc}_A$. Thus $\phi_w^{\mathsf{ref}\ B}(\pi_n^{\mathsf{ref}\ Bw}(x)) = \phi_w^{\mathsf{ref}\ B}(x) = x \in \mathsf{Loc}$, which entails

$$\langle \pi_n^{\mathsf{ref}\ Bw}(x), \phi_w^{\mathsf{ref}\ B}(\pi_n^{\mathsf{ref}\ Bw}(x))\rangle = \langle x, x\rangle \in R_w^{\mathsf{ref}\ B}$$

This concludes this part of the proof.

2. Suppose $\pi_n^{Sw}(s)\downarrow$ and let $s_n = \pi_n^{Sw}(s) = \{\!|\, l_A = \pi_{n-1}^{Aw}(s.l_A) \,|\!\}_{l_A\in w}$, and so

$$\begin{aligned}\phi_w^{St}(s_n) &= \{\!|\, l_A = \phi_w^A(s_n.l_A) \,|\!\}_{l_A\in w}\\ &= \{\!|\, l_A = \phi_w^A(\pi_{n-1}^{Aw}(s.l_A)) \,|\!\}_{l_A\in w}\end{aligned}$$

Then $\mathsf{dom}(s_n) = w = \mathsf{dom}(\phi_w^{St}(s_n))$. Moreover, the first part of the induction hypothesis yields $\langle s_n.l_A, \phi_w^{St}(s_n).l_A\rangle \in R_w^A$, for all $l_A \in w$, i.e., $\langle s_n, \phi_w^{St}(s_n)\rangle \in R_w^{St}$ as required.

3. Again, we consider cases for A:

- Case $A \equiv \mathsf{bool}$: By the definition of R_w^{bool}, $y \in \mathsf{BVal}$ and $x = y$. The result follows immediately from $\pi_n^{\mathsf{bool}w}(x) = x = y = \pi_n^{\mathsf{bool}w}(y) = \pi_n^{\mathsf{bool}w}(\psi_w^{\mathsf{bool}}(y))$.
- Case $A \equiv \{\!|\, \mathsf{m}_i : A_i \,|\!\}$: Suppose $\pi_n^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(x)\downarrow$. In particular, since by definition we have $\pi_n^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(x) = \{\!|\, \mathsf{m}_i = \pi_{n-1}^{A_iw}(x.\mathsf{m}_i) \,|\!\}$ this implies $\pi_{n-1}^{A_iw}(x.\mathsf{m}_i)\downarrow$ for all i. The assumption $\langle x, y\rangle \in R_w^{\{\!|\mathsf{m}_i:A_i\,|\!\}}$ gives $\langle x.\mathsf{m}_i, y.\mathsf{m}_i\rangle \in R_w^{A_i}$ for all i, and so by induction hypothesis

$$\pi_{n-1}^{A_i}(x.\mathsf{m}_i) = \pi_{n-1}^{A_i}(\psi_w^{A_i}(y.\mathsf{m}_i))$$

From this the result follows by calculating

$$\begin{aligned}\pi_n^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(x) &= \{\!|\, \mathsf{m}_i = \pi_{n-1}^{A_iw}(x.\mathsf{m}_i) \,|\!\}\\ &= \{\!|\, \mathsf{m}_i = \pi_{n-1}^{A_i}(\psi_w^{A_i}(y.\mathsf{m}_i)) \,|\!\}\\ &= \pi_n^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(\{\!|\, \mathsf{m}_i = \psi_w^{A_i}(y.\mathsf{m}_i) \,|\!\})\\ &= \pi_n^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(\psi_w^{\{\!|\mathsf{m}_i:A_i\,|\!\}w}(y))\end{aligned}$$

- Case $A \equiv B \Rightarrow B'$: By definition, for all $w' \geq w$, $s \in S_{w'}$ and $u \in [\![B]\!]_{w'}$,

$$\pi_n^{B\Rightarrow B'\,w}(x)_{w'}(s,u) = \begin{cases} \langle w'', \langle \pi_{n-1}^{Sw''}(s'), \pi_{n-1}^{B'w''}(u')\rangle\rangle \\ \qquad \text{if } x_{w'}(\pi_{n-1}^{Sw'}(s), \pi_{n-1}^{Bw'}(u)) = \langle w'', \langle s', u'\rangle\rangle \\ \qquad \text{and } \pi_{n-1}^{Sw''}(s')\downarrow, \pi_{n-1}^{B'w''}(u')\downarrow \\ \text{undefined} \\ \qquad \text{otherwise} \end{cases}$$

and

$$\pi_n^{B\Rightarrow B'\,w}(\psi_w^{B\Rightarrow B'}(y))_{w'}(s,u) = \begin{cases} \langle w'', \langle \pi_{n-1}^{Sw''}(\psi_{w''}^{St}(\sigma)), \pi_{n-1}^{B'w''}(\psi_{w''}^{B'}(\nu))\rangle\rangle \\ \quad \text{if } y(\phi_{w'}^{St}(\pi_{n-1}^{Sw'}(s)), \phi_{w'}^{B}(\pi_{n-1}^{Bw'}(u))) \\ \quad = \langle \sigma, \nu\rangle \text{ and } \mathsf{dom}(\sigma) = w'' \geq w' \\ \quad \text{and } \pi_{n-1}^{Sw''}(\psi_{w''}^{St}(\sigma))\downarrow \\ \quad \text{and } \pi_{n-1}^{B'w''}(\psi_{w''}^{B'}(\nu))\downarrow \\ \text{undefined} \\ \quad \text{otherwise} \end{cases}$$

By the first and second parts of the induction hypothesis we know that for $\hat{s} = \pi_{n-1}^{Sw'}(s)$ and $\hat{u} = \pi_{n-1}^{Bw'}(u)$

$$\langle \hat{u}, \phi_{w'}^{B}(\hat{u})\rangle \in R_{w'}^{B} \quad \text{and} \quad \langle \hat{s}, \phi_{w'}^{St}(\hat{s})\rangle \in R_{w'}^{St}$$

So the assumption $\langle x, y\rangle \in R_w^{B\Rightarrow B'}$ yields

$$x_{w'}(\hat{s},\hat{u}) = \langle w'', \langle s', u'\rangle\rangle \quad \Longleftrightarrow \quad y(\phi_{w'}^{St}(\hat{s}), \phi_{w'}^{B}(\hat{u})) = \langle \sigma, \nu\rangle$$

where $\langle s', \sigma\rangle \in R_{w''}^{St}$ and $\langle u', \nu\rangle \in R_{w''}^{B'}$.
If either $\pi_{n-1}^{Sw''}(s')\uparrow$ or $\pi_{n-1}^{B'w''}(u')\uparrow$ then, by definition, both $\pi_n^{B\Rightarrow B'\,w}(x)_{w'}(s,u)\uparrow$ and $\pi_n^{B\Rightarrow B'\,w}(\psi_w^{B\Rightarrow B'}(y))_{w'}(s,u)\uparrow$. So without loss of generality $\pi_{n-1}^{Sw''}(s')\downarrow$ and $\pi_{n-1}^{B'w''}(u')\downarrow$, and by parts 3 and 4 of the induction hypothesis we obtain

$$\pi_{n-1}^{Sw''}(s) = \pi_{n-1}^{Sw''}(\psi_{w''}^{St}(\sigma)) \quad \text{and} \quad \pi_{n-1}^{B'w''}(\nu) = \pi_{n-1}^{B'w''}(\psi_{w''}^{B'}(\nu))$$

Since this holds for all $w' \geq w$, s and u we have in fact shown $\pi_n^{B\Rightarrow B'\,w}(x) = \pi_n^{B\Rightarrow B'\,w}(\psi_w^{B\Rightarrow B'}(y))$ as required.

- Case $A \equiv \mathsf{ref}\ B$: By definition $\langle x, y\rangle \in R_w^{\mathsf{ref}\ B}$ implies $y \in \mathsf{Loc}_A$ and $x = y$. Similar to the case for bool,

$$\pi_n^{\mathsf{ref}\ Bw}(x) = x = y = \pi_n^{\mathsf{ref}\ Bw}(\psi_w^{\mathsf{ref}\ B}(y))$$

as required.

4. Let $\langle s, \sigma\rangle \in R_w^{St}$, and assume $\pi_n^{Sw}(s)\downarrow$, i.e.

$$\pi_n^{Sw}(s) = \{\!|\, l_A = \pi_{n-1}^{Aw}(s.l_A) \,|\!\}_{l_A \in w}$$

In particular $\pi_{n-1}^{Aw}(s.l_A)\downarrow$ for all $l_A \in w$. By definition of R, $\mathsf{dom}(s) = \mathsf{dom}(\sigma)$ and $\langle s.l_A, \sigma.l_A\rangle \in R_w^A$ for all $l_A \in w$. Thus, part 3 of the induction hypothesis entails (since $\pi_{n-1}^{Aw}(s.l_A)\downarrow$) that $\pi_{n-1}^{Aw}(s.l_A) = \pi_{n-1}^{Aw}(\psi_w^A(\sigma.l_A))$ for all $l_A \in w$ and we obtain

$$\begin{aligned} \pi_n^{Sw}(s) &= \{\!|\, l_A = \pi_{n-1}^{Aw}(s.l_A) \,|\!\}_{l_A \in w} \\ &= \{\!|\, l_A = \pi_{n-1}^{Aw}(\psi_w^A(\sigma.l_A)) \,|\!\}_{l_A \in w} \\ &= \pi_n^{Sw}(\psi_w^{St}(\sigma)) \end{aligned}$$

as required.

This concludes the proof of Lemma 8.6.9. □

Proof of Theorem 8.6.8. For the first part, let $x \in [\![A]\!]_w$. As observed above we have $x = \bigsqcup_n \pi_n^{Aw}(x)$, and in particular $\pi_n^{Aw}(x)\downarrow$ for sufficiently large $n \in \mathbb{N}$. By Lemma 8.6.9,

$$\langle \pi_n^{Aw}(x), \phi_w^A(\pi_n^{Aw}(x))\rangle \in R_w^A$$

for all sufficiently large n. Since this forms an increasing chain in the cpo $[\![A]\!]_w \times \mathsf{Val}$, completeness of R_w^A and continuity of ϕ_w^A shows

$$\begin{aligned}\langle x, \phi_w^A(x)\rangle &= \langle \bigsqcup_n \pi_n^{Aw}(x), \phi_w^A(\bigsqcup_n \pi_n^{Aw}(x))\rangle \\ &= \bigsqcup_n \langle \pi_n^{Aw}(x), \phi_w^A(\pi_n^{Aw}(x))\rangle \ \in R_w^A\end{aligned}$$

as required. The other parts are similar. □

8.7 Coherence of the Intrinsic Semantics

We have now all the parts assembled in order to prove coherence, which proceeds exactly as in (Reynolds 2002b): Suppose $\mathcal{P}_1(\Gamma \rhd e : A)$ and $\mathcal{P}_2(\Gamma \rhd e : A)$ are derivations of the judgement $\Gamma \rhd e : A$. We show that their semantics agree. Let $w \in \mathcal{W}$, $\rho \in [\![\Gamma]\!]_w$ and $s \in S_w$. By Theorem 8.6.8 parts (1) and (2), $\langle \rho, \phi_w^\Gamma(\rho)\rangle \in R_w^\Gamma$ and $\langle s, \phi_w^{St}(s)\rangle \in R_w^{St}$. Hence, by two applications of the Basic Lemma of logical relations, either

$$[\![\mathcal{P}_1(\Gamma \rhd e : A)]\!]_w\, \rho s\uparrow \ \wedge\ [\![e]\!]\,(\phi_w^\Gamma(\rho))(\phi_w^{St}(s))\uparrow \ \wedge\ [\![\mathcal{P}_2(\Gamma \rhd e : A)]\!]_w\, \rho s\uparrow$$

or else there exist w_i, s_i, v_i and σ, v such that

$$\begin{aligned}&[\![\mathcal{P}_1(\Gamma \rhd e : A)]\!]_w\, \rho s = \langle w_1, \langle s_1, v_1\rangle\rangle \\ &\wedge\ [\![e]\!]\,(\phi_w^\Gamma(\rho))(\phi_w^{St}(s)) = \langle \sigma, v\rangle \\ &\wedge\ [\![\mathcal{P}_2(\Gamma \rhd e : A)]\!]_w\, \rho s = \langle w_2, \langle s_2, v_2\rangle\rangle\end{aligned}$$

where $\langle s_i, \sigma\rangle \in R_{w_i}^{St}$ and $\langle v_i, v\rangle \in R_{w_i}^A$, for $i = 1, 2$. The definition of the relation $R_{w_i}^{St}$ entails $w_1 = \mathsf{dom}(\sigma) = w_2$, and by Theorem 8.6.8 parts (3) and (4), $s_1 = \psi_{w_1}^{St}(\sigma) = \psi_{w_2}^{St}(\sigma) = s_2$ and $v_1 = \psi_{w_1}^A(v) = \psi_{w_2}^A(v) = v_2$. We have therefore shown

Theorem 8.7.1 (Coherence)**.** *All derivations of a judgement $\Gamma \rhd e : A$ have the same meaning in the intrinsic semantics.*

Remark 8.7.2. *Observe that the condition of the store case of the logical relation,*

$$\langle s, \sigma\rangle \in R_w^{St} \quad \Longrightarrow \quad \mathit{dom}(\sigma) = w$$

is essential in the above proof. Otherwise we would not be able to conclude $w_1 = w_2$.

Further note that the coherence result *does not hold* if the type annotation A in new_A was removed. In particular, there would then be two different derivations of the judgement

$$x{:}\{\mathsf{m} : \mathsf{bool}\} \rhd \mathsf{new}\ x; \mathsf{true} : \mathsf{bool} \tag{8.8}$$

one without use of subsumption, and one where x is coerced to type $\mathbf{1}$ before allocation. The denotations of these two derivations are *different* (clearly not even the resulting extended worlds are equal). It could be argued that, at least in this particular case, this is a defect of the underlying model: The use of a global store does not reflect the fact that the cell allocated in (8.8) above remains *local* and cannot be accessed by any enclosing program. However, in the general case we do not know if the lack of encapsulation is the only reason preventing coherence for terms without type annotations.

8.8 A PER Model of Higher-Order Storage and Subtyping

In this section we consider two consequences of the preceding technical development in more detail. First, the results are used to obtain an extrinsic semantics over the untyped model, based on partial equivalence relations. Then, we discuss how this relates to the model of Abadi and Leino's logic for objects that was considered in Part II.

8.8.1 Extrinsic PER Semantics

Apart from proving coherence, Reynolds used his analogue of Theorem 8.6.8 to develop an *extrinsic* semantics of types for the (purely applicative) language considered in (Reynolds 2002b). Besides Theorem 8.6.8 this only depends on the Basic Lemma, and we can do exactly the same here. More precisely, the binary relation $\|A\|_w := (R^A_w)^{op} \circ R^A_w$, that is, the relation $\|A\|_w \subseteq \mathsf{Val} \times \mathsf{Val}$ defined by

$$\|A\|_w \stackrel{def}{=} \left\{ \langle u, v\rangle \in \mathsf{Val} \times \mathsf{Val} \;\middle|\; \exists a \in [\![A]\!]_w \,.\, \langle a, u\rangle \in R^A_w \;\wedge\; \langle a, v\rangle \in R^A_w \right\} \tag{8.9}$$

is a partial equivalence relation (per) on Val. Note that a direct proof of transitivity is non-trivial, but it follows easily with part (3) of Theorem 8.6.8: In case of existence, the existentially quantified a in (8.9) is uniquely determined as $a = \psi^A_w(u) = \psi^A_w(v)$.

This definition induces a per $\|w\| \subseteq \mathsf{St} \times \mathsf{St}$ on untyped stores for every $w \in \mathcal{W}$, by $\langle \sigma, \sigma'\rangle \in \|w\|$ iff $\mathsf{dom}(\sigma) = w = \mathsf{dom}(\sigma')$ and $\langle \sigma.l_A, \sigma'.l_A\rangle \in \|A\|_w$ for all $l_A \in w$. The Basic Lemma then shows that the semantics is well-defined on $\|-\|$-equivalence classes, in the sense that if $\Gamma \rhd e : A$ then for all $w \in \mathcal{W}$, for all $\langle \eta, \eta'\rangle \in \|\Gamma\|_w$ and all $\langle \sigma, \sigma'\rangle \in \|w\|$,

$$[\![e]\!]\,\eta\sigma\downarrow \vee [\![e]\!]\,\eta'\sigma'\downarrow \implies \begin{cases} [\![e]\!]\,\eta\sigma = \langle \sigma_1, u\rangle \wedge [\![e]\!]\,\eta'\sigma' = \langle \sigma'_1, u'\rangle \wedge \\ \exists w' \geq w.\, \langle \sigma_1, \sigma'_1\rangle \in \|w'\| \wedge \langle u, u'\rangle \in \|A\|_{w'} \end{cases} \tag{8.10}$$

The resulting per model satisfies some of the expected typed equations: For instance, the records $\{\mathsf{f} = \mathit{true}, \mathsf{g} = \mathit{true}\}$ and $\{\mathsf{f} = \mathit{true}, \mathsf{g} = \mathit{false}\}$ are equal at $\{\mathsf{f} : \mathsf{bool}\}$. Unfortunately, no non-trivial equations involving store hold in this model; in particular, locality and information hiding are not captured. This is no surprise since we work with a global store, and the failure of various desirable equations has already been observed for the underlying typed model (Levy 2002).

However, locality is a fundamental assumption underlying many reasoning principles about programs, such as object and class invariants in object-oriented programming. The work of Reddy and Yang (2004), and Benton and Leperchey (2005), shows how more useful

equivalences can be built in into typed models of languages with storable references. We plan to investigate in how far these ideas carry over to full higher-order store.

We remark that, unusually, the per semantics sketched above does not seem to work over an "untyped" partial combinatory algebra: The construction relies on the partition of the location set $\mathsf{Loc} = \bigcup_A \mathsf{Loc}_A$. In particular, the definition of the pers $\|A\|_w$ depends on this rather arbitrary partition. The amount of type information retained by using typed locations allows us to express the invariance required for references in the presence of subtyping. We have been unable to find a more "semantic" condition. In view of this, the "untyped" model could be viewed simply as a means to an end, facilitating the definition of the logical relation and bracketing maps in order to prove coherence.

Nevertheless, as pointed out to us by Bernhard Reus, the per model may be useful for providing a semantics of languages with *down-casts*, for example in the form of a construct

$$\frac{\Gamma \rhd x : A \quad \Gamma \rhd e_1 : B \Rightarrow C \quad \Gamma \rhd e_2 : A \Rightarrow C}{\Gamma \rhd \mathsf{try}\ (B)x\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 : C}\ (B \preceq A)$$

The intrinsic semantics of Section 8.4 is not suitable for this purpose: For instance, due to the use of coercions, it is impossible to recover "forgotten" fields of a record.

8.8.2 On Abadi and Leino's Logic of Objects

Further, it is interesting to observe the rôle that the typed "witness" of $\langle x_1, x_2\rangle \in \|A\|_w$ plays, i.e., the unique element $a \in [\![A]\!]_w$ with $\langle a, x_i\rangle \in R^A_w$: Crucially, a determines the world $w' \geq w$ over which the result store and value are to be interpreted in the case of application. This is closely related to the soundness proof of the program logic for objects of Chapter 4. Recall that an untyped domain

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

is employed there, representing objects consisting of records of fields $\mathsf{f} \in \mathcal{F}$ and parameterless methods $\mathsf{m} \in \mathcal{M}$, where $S = \mathsf{Rec}_{\mathsf{Loc}}(O)$ denotes object stores. Types (and more generally, specifications) are interpreted as predicates over O. Due to the use of the higher-order store S, types were lifted to a notion of store typing Σ, leading to recursively defined semantics of types as in the present chapter. The semantics of object types in Chapter 4 involves a clause for methods m of type A, similar to (8.10) above. Informally, for all Σ and all Σ-stores σ,

$$m(l, \sigma) = \langle \sigma', v\rangle \implies \exists \Sigma' \geq \Sigma. v \in [\![A]\!]_{\Sigma'} \text{ and } \sigma' \text{ is a } \Sigma'\text{-store} \tag{8.11}$$

where $[\![A]\!]_{\Sigma'}$ is the appropriate denotation of type A. But the use of an existential quantification is problematic. As is well-known, and demonstrated by the example in Section 5.1.2, admissibility is not preserved by arbitrary unions. Thus, existential quantifications preclude the use of the machinery of Pitts (Section 2.5).

In Section 5.1.3 the workaround for this problem was to construct a domain of "choice functions" ϕ that track computations in the untyped model on the level of worlds Σ to provide a witness w. Then, in (8.11) above, the existential quantifier can be replaced by

"$\phi(\Sigma, \sigma) = \Sigma'$ for some $\Sigma' \geq \Sigma$ such that..."

Regarding the setting of the per model in this section, the tracking of the computation on *StSpec* is hard-wired into the witnesses coming from the typed model.

8.8.3 Polymorphism

It is possible to extend the language and the type system with (explicit) predicative, prenex-polymorphism, similar to the (implicit) polymorphism found in Standard ML (Milner, Tofte, Harper, and MacQueen 1997) and Haskell (Peyton Jones 2003). Essentially, the type system is stratified into simple types and *type schemes*, with bounded universally quantified type variables ranging over simple non-polymorphic types only; moreover, the quantification occurs only on top-level:

$$A, B \in \mathit{Type} ::= \ldots \mid \alpha$$

where α ranges over a countably infinite set of type variables, and

$$\sigma \in \mathit{TypeScheme} ::= A \mid \forall \alpha \preceq A.\, \sigma$$

where α is bound in σ by the universal quantifier.

While this form of polymorphic typing may seem fairly restricted, it has proved very popular and useful in practice: It provides a good compromise between expressiveness and type inference that is tractable in many relevant cases, witnessed by the ML and Haskell languages (Milner 1978; Wright 1995).

In (Schwinghammer 2005a) we have shown that the coherence proof can be extended to such a language. However, similar to the per model described in Section 8.8.1 the "untyped" model contains a lot of additional, syntactically defined structure in order to disambiguate the world a store belongs to. We found this necessary in the definition of the bracketing maps. This has the consequence that the model does not exhibit the uniformity generally associated with models of polymorphism; we therefore regard it purely as a technical tool to establish coherence rather than a model explaining parametric polymorphism in the presence of storage.

Chapter 9

A Typed Model of Objects

In this chapter we show that our simple notion of subtyping is useful in obtaining a pleasingly straightforward (typed) semantics of the object calculus (Abadi and Cardelli 1996). We proceed as follows:

Section 9.1 extends the technical results of the previous chapter to also cover a fixed point combinator. Some care is necessary in formalising such combinators in the presence of side-effects. In Section 9.2 a semantics of objects is given, employing a typed variant of the fixed point model of (Reddy 1988; Kamin and Reddy 1994). Section 9.3 illustrates how to prove properties of programs by reasoning about their denotations in the model:

- Reasoning about simple objects, such as the factorial implementation (1.3) from the Introduction, proceeds by a fixed point induction.
- The object-oriented, “circular" implementation of the factorial function (1.4) from the Introduction is proved correct. This requires recursively defined predicates to reason about the use of higher-order store.
- The advantage of using an intrinsically typed model is evidenced by revisiting the simple call-back protocol considered in (Reus and Streicher 2004): A more concise specification of such call-backs is provided. We prove an implementation correct with respect to this specification.
- Abstracting from the examples, an object introduction rule can be proved sound by fixed-point induction. This is a semantic analogue of rule (AL Obj) of Abadi and Leino’s logic.

We explain the failure of our tentative reasoning about (1.2) from the introduction: The specification in this particular instance corresponds to a predicate that is not closed under taking least upper bounds.

9.1 Recursive Functions

As a first step, we show how to interpret explicit recursion in the model, as opposed to recursion through the store due to self-application. Recursive functions will be used in the next section to resolve the dependence of methods on the self parameter.

Call-by-value languages do not provide a fixed-point operator at all types. A common restriction is to add a constant

$$\Gamma \rhd \mathsf{fix}_A : (A \Rightarrow A) \Rightarrow A$$

for functional types $A \equiv B \Rightarrow B'$ only. The existence of fixed points is then guaranteed by the pointedness of (the denotations) of such types in **pCpo** (but see also Boudol's (2004) recent work on a more generous "safe" value recursion). However, in the presence of computational effects there remains the question about the meaning of

$$\Gamma \rhd \mathsf{fix}_{A\Rightarrow B}\,(\lambda f.e) : A \Rightarrow B \tag{9.1}$$

i.e., where the expression e is not necessarily a value. It could be taken to be equivalent to the unfolding $e[(\mathsf{fix}_{A\Rightarrow B}\,\lambda f.e)/f]$, thus duplicating the side-effects of e. An alternative semantics would be to evaluate e to a value v first, and then define (9.1) to be equivalent to $v[(\mathsf{fix}_{A\Rightarrow B}\,\lambda f.v)/f]$. In the latter case the side-effects are performed only once. These issues are discussed in (Erkök and Launchbury 2000; Moggi and Sabry 2004). Here we decided to avoid this problem and follow the simpler approach of the Standard ML language: Essentially, (9.1) is well-formed only if $e \in \mathsf{Val}$ (i.e., e must be an abstraction). We capture this restriction syntactically: The syntax of values is extended by a case for recursive functions rather than providing a fixed point constant,

$$v \in \mathsf{Val} ::= \ldots \mid \mu f(x).e$$

which may be thought of as $\mathsf{fix}_{A\Rightarrow B}\,(\lambda f \lambda x.e)$. We add the new type inference rule

$$\frac{\Gamma, f{:}A\Rightarrow B, x{:}A \rhd e : B}{\Gamma \rhd \mu f(x).e : A \Rightarrow B}$$

As for the semantics, note that each derivation $\mathcal{P}(\Gamma, f{:}A\Rightarrow B, x{:}A \rhd e : B)$ determines a total natural transformation $F : [\![\Gamma]\!] \times [\![A \Rightarrow B]\!] \longrightarrow [\![A \Rightarrow B]\!]$ in $[\mathcal{W}, \mathbf{Cpo}]$, given by

$$F_w(\rho, h) \quad \stackrel{def}{=} \quad \lambda_{w'\geq w}\lambda\langle a, s\rangle.\ [\![\mathcal{P}(\Gamma, f{:}A\Rightarrow B, x{:}A \rhd e : B)]\!]_{w'}\ (([\![\Gamma, f{:}A\Rightarrow B]\!]_w^{w'}\ \rho[f{:=}h])[x{:=}a])s$$

Moreover, every cpo $[\![A \Rightarrow B]\!]_w$ is pointed, with least element $\bot_w$ given by the function that is undefined everywhere, $\bot_w \stackrel{def}{=} \lambda_{w'\geq w}\lambda\langle a, s\rangle\uparrow$. By monotonicity of F_w in each argument,

$$\bot_w \sqsubseteq G_{w\rho}(\bot_w) \sqsubseteq G^2_{w\rho}(\bot_w) \sqsubseteq \ldots$$

forms a chain for all $w \in \mathcal{W}$ and $\rho \in [\![\Gamma]\!]_w$, where $G_{w\rho} \stackrel{def}{=} \lambda h.F_w(\rho, h)$. Hence by continuity of F_w the least fixed point exists in $[\![A \Rightarrow B]\!]_w$,

$$lfp(G_{w\rho}) = \bigsqcup_n G^n_{w\rho}(\bot_w) = G_{w\rho}(lfp(G_{w\rho})) = F_w(\rho, lfp(G_{w\rho}))$$

Further, by induction it follows that

$$[\![A{\Rightarrow}B]\!]_w^{w'} \circ G^n_{w\rho}(\bot_w) \;=\; G^n_{w'[\![\Gamma]\!]_w^{w'}(\rho)}(\bot_{w'})$$

for all $n \in \mathbb{N}$. Thus $[\![A{\Rightarrow}B]\!]_w^{w'} \circ lfp(G_{w\rho}) = lfp(G_{w'[\![\Gamma]\!]_w^{w'}(\rho)})$ and $\rho \mapsto lfp(G_{w\rho})$ is a natural transformation $[\![\Gamma]\!] \longrightarrow [\![A \Rightarrow B]\!]$. Given the notation as above, we now set

$$\left[\!\!\left[\frac{\Gamma, f{:}A{\Rightarrow}B, x{:}A \rhd e : B}{\Gamma \rhd \mu f(x).e : A \Rightarrow B} \right]\!\!\right]_w \rho s \stackrel{def}{=} \langle w, \langle s, lfp(G_{w\rho})\rangle\rangle \in \textstyle\sum_{w' \geq w} S_{w'} \times [\![A{\Rightarrow}B]\!]_{w'}$$

to obtain a semantics for recursive functions in the typed model. In the untyped model, we simply set

$$[\![\mu f(x).e]\!]\,\eta\sigma \stackrel{def}{=} \langle \sigma, lfp(\lambda h.\, [\![\lambda x.e]\!]\,\eta[f := h])\rangle$$

Finally, we turn to the proof of the Basic Lemma, which extends to the case of recursive functions, too.

Proof of Lemma 8.6.7, extended. Let $\langle s, \sigma\rangle \in R_w^{St}$ and $\langle \rho, \eta\rangle \in R_w^{\Gamma}$. We know that by definition,

$$\left[\!\!\left[\frac{\Gamma, f{:}A{\Rightarrow}B, x{:}A \rhd e : B}{\Gamma \rhd \mu f(x).e : A \Rightarrow B} \right]\!\!\right]_w \rho s = \langle w, \langle s, lfp(G_{w\rho})\rangle\rangle$$

and

$$[\![\mu f(x).e]\!]\,\eta\sigma = \langle \sigma, lfp(\lambda h.\, [\![\lambda x.e]\!]\,\eta[f := h])\rangle$$

By assumption, $\langle s, \sigma\rangle \in R_w^{St}$, and it remains to show that the two fixed points are related by $R_w^{A\Rightarrow B}$.

To see this, first observe that $\langle \rho[f := h], \eta[f := h']\rangle \in R_w^{\Gamma, f:A\Rightarrow B}$ for all $\langle h, h'\rangle \in R_w^{A\Rightarrow B}$. Therefore as in the case (Lambda) of non-recursive functions, from the induction hypothesis $\Gamma, f{:}A{\Rightarrow}B, x{:}A \rhd e : B$ it follows that

$$\langle G_{w\rho}(h), [\![\lambda x.e]\!]\,\eta[f := h]\rangle \in R_w^{A\Rightarrow B} \tag{9.2}$$

for all $\langle h, h'\rangle \in R_w^{A\Rightarrow B}$. From the definition of $R_w^{A\Rightarrow B}$ it is immediate that $\langle \bot_w, \bot\rangle \in R_w^{A\Rightarrow B}$ where $\bot = \lambda\langle\sigma, u\rangle\uparrow$ is the everywhere-undefined function in Val. Therefore induction on n and (9.2) shows

$$\langle G_{w\rho}^n(\bot_w), (\lambda h.\, [\![\lambda x.e]\!]\,\eta[f := h])^n(\bot)\rangle \in R_w^{A\Rightarrow B}$$

for all $n \in \mathbb{N}$. Thus, $\langle lfp(G_{w\rho}), lfp(\lambda h.\, [\![\lambda x.e]\!]\,\eta[f := h])\rangle \in R_w^{A\Rightarrow B}$ by admissibility of $R_w^{A\Rightarrow B}$. □

9.2 Objects

Next, we sketch how to give a semantics to Abadi and Cardelli's imperative object calculus with first-order types. As in Chapter 3 we distinguish between fields and methods; fields are mutable, but methods cannot be updated. In fact, we are slightly more general here: Rather than encoding functions as objects, methods can have parameters. The type of objects with fields f_i of type A_i and methods m_j of type C_j (with self parameter y_j) and

parameter z_j of type B_j, is written $[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j \Rightarrow C_j]_{i,j}$. Thus, the introduction rule for objects is

$$\text{(Term Obj)} \qquad \frac{\begin{array}{c} A \equiv [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j \Rightarrow C_j]_{i,j} \\ \Gamma \rhd x_i : A_i \ \forall i \quad \Gamma, y_j{:}A, z_j{:}B_j \rhd b_j : C_j \ \forall j \end{array}}{\Gamma \rhd [\mathsf{f}_i = x_i, \mathsf{m}_j = \varsigma(y_j)\lambda z_j.\, b_j]_{i,j} : A}$$

As in Chapter 3, subtyping on objects is by width, and for methods also by depth, taking the contra-variant position of the method parameter into account:

$$\text{(Sub Obj)} \qquad \frac{B'_j \preceq B_j \quad C_j \preceq C'_j \quad \forall j \in J' \qquad I' \subseteq I \quad J' \subseteq J}{[\mathsf{f}_i : A_i, \mathsf{m}_j : B_j \Rightarrow C_j]_{i\in I, j\in J} \preceq [\mathsf{f}_i : A_i, \mathsf{m}_j : B'_j \Rightarrow C'_j]_{i\in I', j\in J'}}$$

The following is essentially a syntactic presentation of the closure model of objects (Kamin and Reddy 1994, cf. Section 3.2), albeit in a typed setting: Objects of type

$$A \quad \equiv \quad [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j \Rightarrow C_j]_{i,j}$$

are simply interpreted as *records* of the corresponding record type

$$A^* \equiv \{\mathsf{f}_i{:}\mathsf{ref}\ A_i^*, \mathsf{m}_j{:}B_j^* \Rightarrow C_j^*\}_{i,j}$$

Note that the self parameter does not play any part in this type – this is in contrast to functional interpretations of objects, see the discussion in Section 3.2. (For a more detailed overview, refer to Bruce, Cardelli, and Pierce 1999.) In particular, soundness of the subtyping rule (Sub Obj) follows directly because it is derivable in the system of Section 8.3.

In the closure semantics a new object $[\mathsf{f}_i{=}x_i, \mathsf{m}_j{=}\varsigma(y_j)\lambda z_j.\, b_j]_{i,j}$ of type A is created by allocating a state record s and defining the methods by mutual recursion (using obvious syntax sugar),

$$\mathsf{let}\ s = \{\mathsf{f}_i = \mathsf{new}_{A_i}(x_i)\}_{i\in I}\ \mathsf{in}\ \mathit{Meth}_A(s)(\{\mathsf{m}_j = \lambda y_j \lambda z_j.\, b_j\}_{j\in J})$$

where $\mathit{Meth}_A : \{\mathsf{f}_i{:}\mathsf{ref}\ A_i\}_{i\in I} \Rightarrow \{\mathsf{m}_j{:}A^* \Rightarrow B_j \Rightarrow C_j\}_{j\in J} \Rightarrow A^*$ is the recursive function

$$\mathit{Meth}_A \equiv \mu f(s).\lambda m.\ \{\mathsf{f}_i = s.\mathsf{f}_i, \mathsf{m}_j = \lambda z_j.\, (m.\mathsf{m}_j(f(s)(m)))(z_j)\}_{i\in I, j\in J}$$

Soundness of the introduction rule (Term Obj) follows immediately from this interpretation of objects and object types.

The semantics of field selection and field update are simply dereferencing and update, respectively, of the corresponding field of the record. The reduction $(-)^*$ of objects to the procedural language of Section 8.3 is made precise in Table 9.1.

Remark 9.2.1. *So far, we are not able to interpret a cloning construct. Following (Abadi, Cardelli, and Viswanathan 1996) this could be added to the translation of Table 9.1 once we introduce recursive types: The translation of each object of type A would contain a* clone *method of type $\mathbf{1} \Rightarrow A^*$, that allocates new state and then calls the constructor Meth_A as in the object introduction.*

Table 9.1 *Translation of object calculus types and terms*

Types	$[\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j \Rightarrow C_j]^*_{i\in I, j\in J} \equiv \{\mathsf{f}_i{:}\mathsf{ref}\ A_i^*, \mathsf{m}_j{:}B_j^* \Rightarrow C_j^*\}_{i,j}$
Terms	$x^* \equiv x$
	$\mathsf{true}^* \equiv \mathsf{true}$
	$(\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b)^* \equiv \mathsf{if}\ x\ \mathsf{then}\ a^*\ \mathsf{else}\ b^*$
	$(\mathsf{let}\ x = a\ \mathsf{in}\ b)^* \equiv \mathsf{let}\ x = a^*\ \mathsf{in}\ b^*$
	$(a.\mathsf{f})^* \equiv \mathsf{deref}(a.\mathsf{f})$
	$(a.\mathsf{f} := b)^* \equiv (a^*.\mathsf{f}){:=}b^*$
	$(a.\mathsf{m}(b))^* \equiv a^*.\mathsf{m}(b^*)$
	$[\mathsf{f}_i{=}x_i, \mathsf{m}_j{=}\varsigma(y_j)\lambda z_j.\, b_j]^*_{i\in I, j\in J}$
	$\equiv \mathsf{let}\ s = \{\mathsf{f}_i = \mathsf{new}_{A_i}(x_i)\}_{i\in I}\ \mathsf{in}\ \mathit{Meth}_A(s)(\{\mathsf{m}_j = \lambda y_j \lambda z_j.\, b_j^*\}_{j\in J})$
where	$A \equiv [\mathsf{f}_i{:}A_i, \mathsf{m}_j{:}B_j \Rightarrow C_j]_{i\in I, j\in J}$
	$\mathit{Meth}_A \equiv \mu f(s).\lambda m.\, \{\mathsf{f}_i = s.\mathsf{f}_i, \mathsf{m}_j = \lambda z_j.\, (m.\mathsf{m}_j(f(s)(m)))(z_j)\}_{i\in I, j\in J}$

Remark 9.2.2. *The store model of objects induced by the translation $(-)^*$ differs from the one presented in Section 3.3.3 (and used throughout Part II): Before, objects "lived" in the store (in that object creation put the whole object into the store), and were represented by references (i.e., their location in the store) only. But an object itself was not a value. In contrast, objects in this chapter are particular records, and therefore values that are not necessarily kept in the store; heap store is allocated only for the fields. This is closer to the original operational semantics of the imperative object calculus (Abadi and Cardelli 1996, Chap. 10), rather than the one presented in Section 3.3.2.*

By the previous remark, we should really provide an adequacy result with respect to the operational semantics. However, we refrain from doing so here: We expect that the technical results of (Kamin and Reddy 1994) can be adapted (they prove adequacy between the closure model and self-application model for the functional case only). Instead we conclude this section with a brief discussion on why we found this particular store model to be more appropriate in a typed semantics.

Assume we worked with a store model as in the previous chapter, that is, a variable of type A is in fact a reference to a value in the store, semantically an element of $[\![A]\!]$. Since distinct types denote distinct objects of the category, subtype judgements $A \preceq B$ necessarily give rise to coercions from $[\![A]\!]$ to $[\![B]\!]$. Note that coercions "forget" information, for instance, removing the component with label m_2 in the case of judgement $\{\mathsf{m}_1 : A_1, \mathsf{m}_2 : A_2\} \preceq \{\mathsf{m}_1 : A_1\}$: this is necessary since by definition the elements of $[\![\{\mathsf{m}_1 : A_1\}]\!]_w$ are records that simply do not possess a component with label m_2. Now it cannot be sound to apply this coercion directly on values that are kept in the store. The problem is easily seen by considering an example where two "views" on a single value v

of type $\{\mathsf{m}_1 : A_1, \mathsf{m}_2 : A_2\}$ are used,

$$\mathsf{let}\ x{=}v\ \mathsf{in}\ \mathsf{let}\ y{=}x\ \mathsf{in}\ x.\mathsf{m}_2$$

Applying a coercion from $[\![\{\mathsf{m}_1 : A_1, \mathsf{m}_2 : A_2\}]\!]$ to $[\![\{\mathsf{m}_1 : A_1\}]\!]$ at the point where y is declared to equal x, but at the proper supertype $\{\mathsf{m}_1 : A_1\}$, would render the final expression $x.\mathsf{m}_2$ meaningless. In essence this problem is caused by x and y referencing the same store location, and allowing it to be read and written at different types.

On the other hand, by carefully separating between a stored value and its reference in the type system (cf. the type constructor $\mathsf{ref}\ (-)$), the problem described above is avoided: By definition, there is no nontrivial subtyping of reference types.

9.3 Reasoning about Higher-order Store and Objects

Of course, one of the main motivations for devising a denotational semantics is to provide proof principles. A model should enable us to specify the behaviour of, and reason about, concrete programs.

We look at a few small case studies in this section: Firstly, an object with *recursive methods*, in the form of the factorial implementation (1.3) that has been considered in more detail in Chapter 4. The proof principle is fixed point induction. Secondly, *recursion through the store*, exemplified by the object-based implementation of the factorial function (1.4), where the recursion is resolved by calling the method through an object stored in a member field. This calls for recursively defined predicates whose well-definedness has to be established first, similar to the existence proof for the Kripke logical relation of Section 8.6.

Next, we consider a simple call-back mechanism (Gamma, Helm, Johnson, and Vlissides 1995): the method cb we wish to specify simply sends requests on to a method m of an object accessible via one of its fields f. As such, this method may be changed at run-time. To reflect this, a sensible specification of the call-back would be of the form

if method m *satisfies a specification* S*, then* S *holds of* cb *too*

where S ranges over a suitable class of specifications.

We conclude by considering a proof rule for reasoning about the introduction of objects, similar to the object introduction rule (AL OBJ) of Table 4.4 on page 71.

9.3.1 Recursive Methods: The Factorial

In the first example we proceed by using fixed point induction. Recall the object implementing the factorial from Chapter 4:

$$a \equiv \left[\begin{array}{l} \mathsf{arg} = 0, \\ \mathsf{fac} = \varsigma(y)\lambda(z{:}\mathbf{1}).\ \mathsf{if}\ y.\mathsf{arg} = 0\ \mathsf{then}\ 1 \\ \qquad\qquad\qquad \mathsf{else\ let}\ x{=}y.\mathsf{arg}\ \mathsf{in}\ y.\mathsf{arg}{:=}x - 1; x \times (y.\mathsf{fac}()) \end{array}\right]$$

There, we used the proof rules of Abadi and Leino's logic to show that the program satisfies the specification A, where

$$A \equiv [\mathsf{arg}:\mathsf{int},\ \mathsf{fac}:\varsigma(y)\mathsf{int}::\mathsf{sel}_{pre}(y,\mathsf{arg}) \geq 0 \ \rightarrow\ \mathsf{result} = \mathsf{sel}_{pre}(y,\mathsf{arg})!]$$

Here, we show this in an ad-hoc way by reasoning directly about the denotation of a. Thus suppose w_0 is some world containing at least a location $l \in \mathsf{Loc}_{\mathsf{int}}$, let B stand for $[\mathsf{arg}:\mathsf{int},\ \mathsf{fac}:\mathbf{1}\Rightarrow\mathsf{int}]^*$, and let $\mathcal{A} = (A_w)_{w\geq w_0}$ be the family of predicates $A_w \subseteq [\![B]\!]_w$ defined by

$$\begin{array}{rcl} o \in A_w & \stackrel{def}{\Longleftrightarrow} & \forall w'' \geq w' \geq w\ \forall s \in S_{w'}\ \forall s' \in S_{w''}\, \forall n \in \mathbb{N}. \\ & & \quad s.(o.\mathsf{arg}) \geq 0\ \wedge\ o.\mathsf{fac}_{w'}(s, \{\!\}) = \langle w'', \langle s', n\rangle\rangle\ \Longrightarrow\ n = (s.(o.\mathsf{arg}))! \end{array}$$

Recall that the denotation of an object is determined by application of the recursively defined constructor $Meth_A$ to the state record and record of pre-methods. We observe that for all w the pre-method $h \stackrel{def}{=} [\![\triangleright\lambda y\lambda z.\mathsf{if}\ \mathsf{deref}(y.\mathsf{arg})=0\ \mathsf{then}\ 1\ \mathsf{else}\ e]\!]_w$, where e is $\mathsf{let}\ x=\mathsf{deref}(y.\mathsf{arg})\ \mathsf{in}\ y.\mathsf{arg}:=x-1;\ x\times y.\mathsf{fac}()$, satisfies

$$\begin{array}{r} \forall w_2 \geq w_1 \geq w\ \forall s \in S_{w_1}\ \forall o \in A_{w_1}\ \forall s' \in S_{w_2}\ \forall n \in [\![\mathsf{int}]\!]_{w_2}.\quad h_{w_1}(s,o) \in [\![\mathbf{1}\Rightarrow\mathsf{int}]\!]_{w_1}\ \wedge \\ s.(o.\mathsf{arg}) \geq 0\ \wedge\ (h_{w_1}(s,o))_{w_1}(s,\{\!\}) = \langle w_2, \langle s', n\rangle\rangle\ \Longrightarrow\ n = s.(o.\mathsf{arg})! \end{array} \quad (9.3)$$

Next consider the admissible predicate $M_w \subseteq [\![\{\mathsf{arg}:\mathsf{ref}\ \mathsf{int}\}\Rightarrow\{\mathsf{fac}:B\Rightarrow\mathbf{1}\Rightarrow\mathsf{int}\}\Rightarrow B]\!]_w$,

$$\begin{array}{rcl} \mu \in M_w & \stackrel{def}{\Longleftrightarrow} & \forall s \in S_w.\ \mu_w(s,\{\!|\mathsf{arg}=l|\!\})\downarrow\ \Longrightarrow\ \exists\mu'.\ \mu_w(s,\{\!|\mathsf{arg}=l|\!\}) = \langle w, \langle s, \mu'\rangle\rangle \\ & & \wedge\ \mu'_w(s,\{\!|\mathsf{fac}=h|\!\})\downarrow\ \Longrightarrow\ \exists o.\ \mu'_w(s,\{\!|\mathsf{fac}=h|\!\}) = \langle w, \langle s, o\rangle\rangle\ \wedge\ o \in A_w \end{array}$$

By fixed point induction, using (9.3),

$$[\![\triangleright Meth_B : \{\mathsf{arg}:\mathsf{ref}\ \mathsf{int}\}\Rightarrow\{\mathsf{fac}:B\Rightarrow\mathbf{1}\Rightarrow\mathsf{int}\}\Rightarrow B]\!]_w \in M_w$$

Thus,

$$\begin{array}{rl} [\![\triangleright a:B]\!]_w\, s = & \mathbf{let}\ \langle w', \langle s', \mu'\rangle\rangle = ([\![\triangleright Meth_B : \ldots]\!]_w\, s)_w(s+\{\!|l=0|\!\}, \{\!|\mathsf{arg}=l|\!\}) \\ & \mathbf{in}\ \mu'_w(s', \{\!|\mathsf{fac}=h|\!\}) \in S_w \times A_w \end{array}$$

follows by definition of M_w.

9.3.2 Recursion through the Store: The Factorial

In the following program let $A \equiv [\mathsf{fac}:\mathsf{int}\Rightarrow\mathsf{int}]$, and $B \equiv [\mathsf{f}:A, \mathsf{fac}:\mathsf{int}\Rightarrow\mathsf{int}]$ (so that $B \preceq A$ holds). The program also computes the factorial, but making the recursive calls through the store. Suppose x is declared as an integer variable, and consider the program

$$\begin{array}{l} \mathsf{let}\ a:A = [\mathsf{fac} = \varsigma(x)\lambda n.\ n]\ \mathsf{in} \\ \mathsf{let}\ b:B = [\mathsf{f} = a, \mathsf{fac} = \varsigma(x)\lambda n.\ \mathsf{if}\ n<1\ \mathsf{then}\ 1\ \mathsf{else}\ n\times(x.\mathsf{f}.\mathsf{fac}(n-1))] \\ \mathsf{in}\ b.\mathsf{f} := b;\ b.\mathsf{fac}(x) \end{array}$$

While we certainly do not claim that this is a particularly realistic example, it does show how higher-order store complicates reasoning. We illustrate a pattern for dealing with this form of self-application, arising from the use of higher-order store, following the general ideas of (Reus and Streicher 2004) and (Pitts 1996): To prove that the call in the last line indeed computes the factorial of x, consider the family of predicates $P = (P_w)_w$, where w ranges over worlds $\geq \{l{:}A\}$ and $P_w \subseteq [\![\mathsf{int} \Rightarrow \mathsf{int}]\!]_w$,

$$h \in P_w \quad \overset{def}{\iff} \quad \forall w' \geq w\ \forall s \in S_{w'}\ \forall n \in [\![\mathsf{int}]\!]_{w'}.\ (s.l.\mathsf{fac} \in P_{w'} \wedge n \geq 0 \wedge h_{w'}(s,n)\!\downarrow)$$
$$\Longrightarrow\ \exists w'' \geq w'\ \exists s' \in S_{w''}.\ h_{w'}(s,n) = \langle w'', \langle s', n!\rangle\rangle$$

Note that P_w corresponds to a partial correctness assertion, i.e., *if* the result is defined, then it is indeed $n!$. This example has also been considered in the context of total correctness, in recent work of Honda *et al.* (Honda, Yoshida, and Berger 2005) (where, rather different to here, the proof relies on well-founded induction using a termination order).

Due to the (negative) occurrence of $P_{w'}$ in the definition of P_w existence of such a family P has to be established. This can be done along the lines of Theorem 8.6.4: A relational structure $\mathcal{R}$ on the category C is given by defining $\mathcal{R}(X)$ to be the type- and world-indexed admissible relations on X, and defining

$$f : R \subset T \quad \overset{def}{\iff} \quad \begin{cases} \forall w \in \mathcal{W}\ \forall A \in \mathit{Type}\ \forall x \in R^A_w.\ f_{Aw}(x)\!\downarrow\ \Longrightarrow\ f_{Aw}(x) \in T^A_w \\ \forall w \in \mathcal{W}\ \forall s \in R^{St}_w.\ f_{Sw}(s)\!\downarrow\ \Longrightarrow\ f_{Sw}(s) \in T^{St}_w \end{cases}$$

for all $R \in \mathcal{R}(X)$, $T \in \mathcal{R}(Y)$ and C-morphisms $f : X \to Y$. A functional Φ is defined corresponding to the predicate $\mathcal{P}$ above,

$$f \in \Phi(R)^{\mathsf{int}\Rightarrow\mathsf{int}}_w \ \overset{def}{\iff}\ \forall w'' \geq w' \geq w\ \forall n \geq 0\ \forall s \in S_{w'}\ \forall m \in [\![\mathsf{int}]\!]_{w''}\ \forall s' \in S_{w''}.$$
$$(s.l \in R^{\mathsf{int}\Rightarrow\mathsf{int}}_{w'} \wedge f_{w'}(s,n) = \langle w'', \langle s', m\rangle\rangle\ \Longrightarrow\ m = n!)$$

at worlds $w \geq \{l{:}\mathsf{int} \Rightarrow \mathsf{int}\}$ (the value of Φ at other types, as well as on worlds not extending $\{l{:}\mathsf{int} \Rightarrow \mathsf{int}\}$, does not really matter and could be chosen as the empty relation, for instance). This definition forms an admissible action of the functor $F : C \to C$ used to construct the model:

$$e^- : R' \subset R \ \wedge\ e^+ : T \subset T' \quad \Longrightarrow \quad F(e^-, e^+) : \Phi(R) \subset \Phi(R') \tag{9.4}$$

As in Section 8.6, property (9.4) suffices to establish well-definedness of the predicates P, by Theorem 2.5.2.

Assuming that l is the location allocated for field f, a fixed-point induction shows

$$[\![x{:}\mathsf{int}, a{:}A \rhd [\mathsf{f} = a, \mathsf{fac} = \varsigma(x)\lambda n.\ \mathsf{if}\ \ldots\ ] : B]\!]_w\, \rho s = \langle w', \langle s', o\rangle\rangle \tag{9.5}$$

such that w' is $w \cup \{l{:}A\}$, and $o.\mathsf{fac} \in P_{w'}$.

Now let $\hat{s} = s'[l := [\![B \preceq A]\!]_{w'}(o)]$. Thus, $\hat{s}.l.\mathsf{fac} = o.\mathsf{fac} \in P_{w'}$; and if $\rho(x) \geq 0$ we conclude

$$[\![x{:}\mathsf{int}, a{:}A, b{:}[\mathsf{f}{:}A,\ \mathsf{fac}{:}\mathsf{int}\Rightarrow\mathsf{int}] \rhd b.\mathsf{f} := b; b.\mathsf{fac}(x) : \mathsf{int}]\!]_{w'}\, \rho[b := o]\hat{s}$$
$$= \hat{s}.l.\mathsf{fac}_{w'}(\hat{s}, \rho(x))$$
$$= \langle w'', \langle s'', \rho(x)!\rangle\rangle$$

for some w'' and s''.

Remark 9.3.1. *In the case of the object calculus, subtyping seems necessary to construct such a recursion through the store; otherwise (in the absence of recursive types) one could not assign the object itself to one of its fields. It is even easier (in the sense that neither recursive types nor subtyping are necessary) to construct similar examples in the calculus of Section 8.3 where references on their own are an integral part of the language, rather than being available only as part of an object. One such example was* (1.2) *in the Introduction.*

9.3.3 Call-backs

As another example, we treat the call-back example considered in (Reus and Streicher 2004). Call-backs are used in object-oriented programming to decouple the dependency between caller and callee objects. A typical example is that of generic buttons in user interface libraries, described in (Gamma, Helm, Johnson, and Vlissides 1995) by the *command pattern*: As the implementor of the button class cannot have any knowledge about the functionality associated with a particular window button instance, it is assumed that there will be an object supplied (at run-time) that encapsulates the desired behaviour for the *button pressed* event, by providing a method `execute`. That is, the object should have type

$$[\mathsf{execute} : \mathbf{1} \Rightarrow \mathbf{1}]$$

Apart from implementing this interface, there are no further requirements on the supplied object. In particular, no assumptions about the behaviour of its `execute` method are made. The `buttonPressed` method of the button class will then react to events by forwarding to the `execute` method. In terms of specifications, `buttonPressed` would thus satisfy any specification that `execute` satisfies.

The techniques developed in (Reus and Streicher 2004) to express such parametric specifications carry over to the present semantics. However, in contrast to *loc. cit.*, and highlighting the fact that our model is typed, we are able to give a more concise specification of such call-back methods (see (9.7) and (9.8) below): The existence of all the required methods in the participating objects is already ensured by the intrinsic typing. In the untyped semantics of (Reus and Streicher 2004), existence of a method named `execute` has to be required explicitly in the specification of the method `buttonPressed`. The price we pay for this is that generally we must consider *families* of predicates, indexed by worlds and types.

We proceed analogously to (Reus and Streicher 2004). Before considering the button example in detail, a notation for semantic Hoare triples is defined.

Definition 9.3.2. *For $X \in [\mathcal{W}, \mathbf{pCpo}]$ let $\mathcal{A}dm(X)$ denote the complete lattice of families $\mathcal{P} = (P_w)_{w\in\mathcal{W}}$ of admissible predicates $P_w \subseteq S_w \times X_w$, ordered pointwise by inclusion.*

Now suppose $\mathcal{P} \in \mathcal{A}dm([\![A]\!])$ and $\mathcal{Q} \in \mathcal{A}dm([\![B]\!])$ are such families of predicates. For $w \in \mathcal{W}$ and $h \in [\![A \Rightarrow B]\!]_w$ we define

$$\begin{aligned}\{\mathcal{P}\}h\{\mathcal{Q}\} \quad \overset{def}{\iff} \quad & \forall w'' \geq w' \geq w\ \forall \langle s, a\rangle \in S_{w'} \times [\![A]\!]_{w'}\ \ \forall \langle s', b\rangle \in S_{w''} \times [\![B]\!]_{w''}\,. \\ & (\langle s, a\rangle \in P_{w'}\ \wedge\ h_{w'}(s, a) = \langle w'', \langle s', b\rangle\rangle\ \Longrightarrow\ \langle s', b\rangle \in Q_{w''})\end{aligned}$$

expressing a partial correctness assertion of the function h. It can be verified that since each Q_w is an admissible subset of $S_w \times [\![B]\!]_w$ then $\{\mathcal{P}\}-\{\mathcal{Q}\}$ denotes an admissible subset of the function space $[\![A \Rightarrow B]\!]_w$.

Lemma 9.3.3. *$\{\mathcal{P}\} - \{\mathcal{Q}\}$ forms an admissible subset of $[\![A \Rightarrow B]\!]$.*

Next we consider an implementation of the generic buttons described above. Let B describe the interface $B \equiv [\mathsf{execute} : \mathbf{1}\Rightarrow\mathbf{1}]$ and let

$$A \equiv [\mathsf{evHdl} : B, \mathsf{buttonPressed} : \mathbf{1}\Rightarrow\mathbf{1}]$$

be the type of such button objects. Let a be the object

$$x{:}B \;\triangleright\; \left[\begin{array}{l} \mathsf{evHdl} = x, \\ \mathsf{buttonPressed} = \varsigma(y)\lambda z.\; y.\mathsf{evHdl}.\mathsf{execute}(\{\}) \end{array}\right] : A \tag{9.6}$$

Let $l \in \mathsf{Loc}_B$, let $w' \geq w \geq \{l{:}B\}$ and set $T^l_{w,w'} \subseteq S_w \times S_{w'}$ to

$$\begin{array}{rcl} \langle s, s'\rangle \in T^l_{w,w'} & \stackrel{def}{\Longleftrightarrow} & \forall \mathcal{P}, \mathcal{Q} \in \mathcal{A}dm([\![\mathbf{1}]\!]). \\ & & \{\mathcal{P}\}s.l.\mathsf{execute}\{\mathcal{Q}\} \;\wedge\; \langle s, \{\!\}\rangle \in P_w \;\Longrightarrow\; \langle s', \{\!\}\rangle \in Q_{w'} \end{array} \tag{9.7}$$

Since each $Q_{w'}$ is admissible and admissible predicates are closed under universal quantification, $T^l_{w,w'}(s,-)$ determines an admissible predicate over $S_{w'}$ for fixed $s \in S_w$. Thus, the set

$$\left\{ h \in [\![\mathbf{1} \Rightarrow \mathbf{1}]\!]_w \;\middle|\; \forall s, s'.\, h_w(s,\{\}) = \langle w', \langle s', \{\}\rangle\rangle \;\Longrightarrow\; \langle s, s'\rangle \in T^l_{w,w'} \right\} \tag{9.8}$$

is admissible in $[\![\mathbf{1} \Rightarrow \mathbf{1}]\!]_w$. Now assume that l is the location allocated for the field evHdl of a. A fixed-point induction shows that for any w and $w' = w \cup \{l{:}B\}$, there is an $o \in [\![A]\!]_{w'}$ such that

$$\langle w', \langle s', o\rangle\rangle = [\![x{:}B \triangleright a : A]\!]_w\, \rho s \tag{9.9}$$

and $o.\mathsf{buttonPressed} \in [\![\mathbf{1} \Rightarrow \mathbf{1}]\!]_{w'}$ satisfies the specification (9.8).

Limits of the Approach

We remark that while this approach goes a long way towards reasoning about call-backs, it reaches its limits when combined with recursion through the store. For example, suppose in the call-back example methods $\mathsf{execute}$ and $\mathsf{buttonPressed}$ had the same name, m, say. Thus, the subtype relation

$$A \;\equiv\; [\mathsf{evHdl} : B,\; \mathsf{m} : \mathbf{1}\Rightarrow\mathbf{1}] \;\preceq\; [\mathsf{m} : \mathbf{1}\Rightarrow\mathbf{1}] \;\equiv\; B$$

between types A and B holds, and the program

$$x{:}B \triangleright \mathsf{let}\; y{=}a \;\mathsf{in}\; y.\mathsf{evHdl}{:=}y;\; y \tag{9.10}$$

is type-correct, where a is as in (9.6). Our intuition tells us that the meaning of any call $y.\mathsf{m}(\{\})$ is undefined (operationally, it diverges) since m immediately calls itself recursively, through the store.

Writing o for the semantics of (9.10) it follows as before (substituting the altered method name m appropriately) that $o.\mathsf{m}$ satisfies the specification (9.8). Expanding the definition, this means for all stores $s \in S_w$ and $s' \in S_{w'}$, and for all $\mathcal{P}, \mathcal{Q} \in \mathcal{A}dm([\![\mathbf{1}]\!])$,

$$\{\mathcal{P}\}s.l.\mathsf{m}\{\mathcal{Q}\} \ \wedge \ o.\mathsf{m}_w(s, \{\!|\,|\!\}) = \langle w', \langle s', \{\!|\,|\!\}\rangle\rangle \ \wedge \ \langle s, \{\!|\,|\!\}\rangle \in P_w \implies \langle s', \{\!|\,|\!\}\rangle \in Q_{w'}$$

Non-termination says there is *no* s' such that $o.\mathsf{m}_w(s, \{\!|\,|\!\}) = \langle w', \langle s', \{\!|\,|\!\}\rangle\rangle$, so we should instantiate $\mathcal{Q}$ above with the family of predicates $\mathcal{F} = (F_w)$ where $F_w \stackrel{def}{=} \emptyset$. But this leaves us with the task of showing the precondition $\{\mathcal{P}\}s.l.\mathsf{m}\{\mathcal{F}\}$ holds, and since $s.l.\mathsf{m} = o.\mathsf{m}$ we must in fact prove $\{\mathcal{P}\}o.\mathsf{m}\{\mathcal{F}\}$. Note that we set out to prove this in the first place!

In fact, the observation that co-inductively defined specifications fail in this instance motivated the study of the more powerful, mixed-variant predicates in (Reus and Streicher 2004): It is observed there that a number of constructions, like $\{\mathcal{P}\} - \{\mathcal{Q}\}$, respect the relational structure (in the sense that specifications built up from basic building blocks satisfy conditions such as (9.4)), therefore possess unique fixed points.

Note that in Chapter 5 such mixed-variant specifications were already necessary to prove the object introduction rule sound, cf. the definition of $[\![\Sigma]\!]$. In comparison, object introduction in the typed semantics requires only fixed point induction, as shown next. The reason is of course that we have replaced the self-application model of objects of Chapter 3 by the fixed point model.

9.3.4 A Semantic Object Introduction Rule

As seen in the previous examples, the introduction of an object usually involves a fixed point induction in order to establish properties of the fields and methods, given that the initial values and pre-methods satisfy given specifications (see (9.5) and (9.9)). This is due to the construction of objects by an application of the constructor $Meth_A$ to the state record and record of pre-methods. We present a semantic rule that encapsulates this fixed point induction.

Let $\mathcal{A}dm$ be as in Definition 9.3.2. Given families of admissible predicates $\mathcal{P}_i \in \mathcal{A}dm([\![A_i]\!])$ for $i \in I$ and $\mathcal{Q}_j \in \mathcal{A}dm([\![B_j]\!])$, $\mathcal{R}_j \in \mathcal{A}dm([\![C_j]\!])$ for $j \in J$, let the family of predicates $Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{i\in I, j\in J}$ where

$$Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_w \ \subseteq \ S_w \times [\![[\mathsf{f}_i : A_i, \, m_j : B_j \Rightarrow C_j]]\!]$$

defined by

$$\langle s, o\rangle \in Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_w \quad \stackrel{def}{\iff} \quad \forall i \in I.\ \langle s, s.(o.\mathsf{f}_i)\rangle \in P_{iw} \ \wedge \ \forall j \in J.\ \{\mathcal{Q}_j\}o.\mathsf{m}_j\{\mathcal{R}_j\}$$

Stability

The following concept identifies a condition on families of admissible predicates that suffices for the proof rule below.

Definition 9.3.4 (Stability)**.** *We call a family of predicates $\mathcal{P} \in \mathcal{A}dm(X)$* stable (under extensions) *if and only if, for all $w \in \mathcal{W}$, all $\langle s, x\rangle \in P_w$ and all $w' \geq w$ and $s' \in S_{w'}$*

$$\forall l_A \in w.\ s'.l = [\![A]\!]^{w'}_w\,(s.l) \quad \implies \quad \langle s', X^{w'}_w(x)\rangle \in P_{w'}$$

That is, the allocation of further locations (with arbitrary contents) does not invalidate the specification, as long as the original part of the store remains unchanged. Furthermore, for $\mathcal{P} \in \mathcal{A}dm([\![A]\!])$ and $\mathcal{Q} \in \mathcal{A}dm([\![B]\!])$ we define $\mathcal{P} \Rightarrow \mathcal{Q} \in \mathcal{A}dm([\![A \Rightarrow B]\!])$ by

$$\langle s, h\rangle \in (\mathcal{P} \Rightarrow \mathcal{Q})_w \quad \stackrel{def}{\iff} \quad \{\mathcal{P}\}h\{\mathcal{Q}\}$$

for $\langle s, h\rangle \in S_w \times [\![A \Rightarrow B]\!]_w$. We observe that $\mathcal{P} \Rightarrow \mathcal{Q}$ is a stable family, since for all $w'' \geq w' \geq w$ we have $([\![A \Rightarrow B]\!]_w^{w'}(h))_{w''} = h_{w''}$ which entails $\{\mathcal{P}\}\,[\![A \Rightarrow B]\!]_w^{w'}(h)\{\mathcal{Q}\}$ whenever $\{\mathcal{P}\}h\{\mathcal{Q}\}$. Similarly, if each $\mathcal{P}_i \in \mathcal{A}dm([\![A_i]\!])$ is stable then so is $Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{i\in I, j\in J}$ for the same reason.

Object Introduction

We introduce the following convenient notation. If $\mathcal{P} = (P_w)_w \in \mathcal{A}dm(X)$ and $r \in \sum_w S_w \times X_w$ we simply write $r \in \mathcal{P}$, meaning that $\langle s, x\rangle \in P_{w'}$ whenever $r = \langle w', \langle s, x\rangle\rangle$.

Now suppose $A \equiv [\mathsf{f}_i : A_i, \mathsf{m}_j : B_j \Rightarrow C_j]_{i\in I, j\in J}$. For all $\mathcal{P}_i \in \mathcal{A}dm([\![A_i]\!])$, $\mathcal{Q}_j \in \mathcal{A}dm([\![B_j]\!])$ and $\mathcal{R}_j \in \mathcal{A}dm([\![C_j]\!])$ the following rule is sound:

$$\frac{\begin{array}{l}\forall i \in I \quad [\![\Gamma \rhd x_i : A_i]\!]_w\, \rho s \in \mathcal{P}_i \\ \forall j \in J \quad [\![\Gamma \rhd \lambda y_j \lambda z_j . b_j : A \Rightarrow B_j \Rightarrow C_j]\!]_w\, \rho s \in Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{i\in I, j\in J} \Rightarrow \mathcal{Q}_j \Rightarrow \mathcal{R}_j\end{array}}{\left[\!\!\left[\Gamma \rhd \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)\lambda z_j . b_j\right]_{i\in I, j\in J} : A\right]\!\!\right]_w \rho s \in Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{i\in I, j\in J}} \tag{9.11}$$

provided each $\mathcal{P}_i$ is stable.

For the proof, assume $\{1, \ldots, n\} = I$, let $w' \stackrel{def}{=} w, l_1{:}A_1, \ldots, l_n{:}A_n$, let $v_i \in [\![A_i]\!]_w$ be such that $[\![\Gamma \rhd x_i : A_i]\!]_w\, \rho s = \langle w, \langle s, v_i\rangle\rangle$ and let

$$s' \stackrel{def}{=} \{\!|l = [\![A]\!]_w^{w'}(s.l)|\!\}_{l_A \in w} + \{\!|l_i = [\![A_i]\!]_w^{w'}(\rho(x_i))|\!\}_{i\in I}$$

so that $\langle w', \langle s', \{\!|\mathsf{f}_i = l_i|\!\}\rangle\rangle = [\![\{\mathsf{f}_i = \mathsf{new}_{A_i}(x_i)\}]\!]_w\, \rho s \in \sum_{w'\geq w} S_{w'} \times [\![\{\mathsf{f}_i : A_i\}]\!]_{w'}$. Next consider the admissible set $M \subseteq [\![\{\mathsf{f}_i : \mathsf{ref}\ A_i\} \Rightarrow \{\mathsf{m}_j : A \Rightarrow B_j \Rightarrow C_j\} \Rightarrow A]\!]_{w'}$,

$$\begin{array}{lll}\mu \in M & \stackrel{def}{\iff} & \mu_{w'}(s', \{\!|\mathsf{f}_i = l_i|\!\})\downarrow \implies \exists \mu'.\ \mu_{w'}(s', \{\!|\mathsf{f}_i = l_i|\!\}) = \langle w', \langle s', \mu'\rangle\rangle \\ & & \wedge\ \mu'_{w'}(s', \{\!|\mathsf{m}_j = h_j|\!\})\downarrow \implies \exists \hat{o}.\ \mu'_{w'}(s', \{\!|\mathsf{m}_j = h_j|\!\}) = \langle w', \langle s', \hat{o}\rangle\rangle \\ & & \wedge\ \langle s', \hat{o}\rangle \in Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{w'}\end{array} \tag{9.12}$$

where h_j is determined as $\langle w', \langle s', h_j\rangle\rangle = [\![\Gamma \rhd \lambda y_j \lambda z_j . b_j : A \Rightarrow B_j \Rightarrow C_j]\!]_{w'}\, \rho s'$. The import of the first two lines of (9.12) is to express that application of μ does not change the store.

Recall the object constructor, $Meth_A$,

$$Meth_A \equiv \mu f(s).\lambda m.\ \{\mathsf{f}_i = s.\mathsf{f}_i, \mathsf{m}_j = \lambda z_j.\ (m.\mathsf{m}_j(f(s)(m)))(z_j)\}_{i\in I, j\in J}$$

We show that (its denotation) is in M. Clearly $\perp \in M$ for $\perp$ the function that is undefined everywhere. Next suppose $\mu \in M$, and let

$$o \stackrel{def}{=} \{\!|\mathsf{f}_i = v_i,\ \mathsf{m}_j = h_{j\,w'}(s', \hat{o})|\!\}_{i,j}$$

where $\hat{o}$ is the result of applying μ to $\langle s', \{\!|\mathsf{f}_i = v_i|\!\}\rangle$ and $\langle s', \{\!|\mathsf{m}_j = h_j|\!\}\rangle$, as in (9.12). By stability, $\langle s', v_i\rangle \in P_{i w'}$ for all $i \in I$. Furthermore, since $\mu \in M$, $\langle s', \hat{o}\rangle \in Ob(\mathcal{P}_i, \mathcal{Q}_j, \mathcal{R}_j)_{w'}$.

Hence, $\{\mathcal{Q}_j\}h_{j\,w'}(s',\hat{o})\{\mathcal{R}_j\}$ from the premiss of the rule, and $\langle s', o\rangle \in Ob(\mathcal{P}_i,\mathcal{Q}_j,\mathcal{R}_j)_{w'}$ follows. By fixed point induction we can therefore conclude

$$[\![\Gamma \rhd \mathit{Meth}_A : \{\mathsf{f}_i : \mathsf{ref}\ A_i\} \Rightarrow \{\mathsf{m}_j : B_j \Rightarrow C_j\} \Rightarrow A]\!]_{w'}\ \rho s' \in M$$

and in particular

$$\begin{aligned}&\left[\!\!\left[\Gamma \rhd \left[\mathsf{f}_i = x_i,\ \mathsf{m}_j = \varsigma(y_j)\lambda z_j.b_j\right]_{i\in I, j\in J} : A\right]\!\!\right]_w \rho s\\ &\quad = \mathbf{let}\ \langle w', \langle s', \mu'\rangle\rangle = ([\![\Gamma \rhd \mathit{Meth}_A : \ldots]\!]_{w'}\ \rho s')_{w'}(s', \{\!|\mathsf{f}_i = v_i|\!\}) \ \mathbf{in}\ \mu'_{w'}(s', \{\!|\mathsf{m}_j = h_j|\!\})\end{aligned}$$

proves the conclusion of the introduction rule (9.11), by definition of M.

Comparison to the work of Reus and Streicher

As indicated by this object introduction rule, the definition of $Ob(\mathcal{P}_i,\mathcal{Q}_j,\mathcal{R}_j)_{w'}$ and fixed point induction are in fact sufficient to prove correctness of Abadi and Leino's object introduction rule (AL Obj). This is in apparent contradiction to (Reus and Streicher 2004) where it was argued that predicates corresponding to specifications of the logic need be mixed-variant. Of course the answer is that Reus and Streicher use the self-application model whereas the fixed point model is used in this chapter. And while the program

$$\mathsf{let}\ x = [\mathsf{m} = \varsigma(y)\lambda z.\, y.\mathsf{m}(z)]\ \mathsf{in}\ x.\mathsf{m}()$$

can be proved non-terminating by rule (9.11), for the variation (9.10) which uses recursion through the store it is not obvious how to proceed *without* mixed-variant specifications. Our, not very formal, conclusion is the following: Since mixed-variant predicates are the proof principle corresponding to recursion through the store, such predicates defined by mixed-variant recursion must be used *somewhere* in order to establish soundness of reasonably expressive (syntactic) proof calculi. In particular, even though reasoning about object construction in the fixed point model proceeds by fixed point induction, a potential recursion through the store may sneak in through field updates. Our previous examples have demonstrated this.

9.3.5 Non-Existence of Specifications

The failed reasoning about program (1.2) in the Introduction shows that predicates satisfying arbitrary recursively defined requirements need not exist. Recall that we attempted to prove for the program

$$\begin{aligned}&\mathsf{let}\ f : \mathsf{int} \Rightarrow \mathsf{int} = \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ (\mathsf{deref}\ r)(n-1)\\ &\mathsf{in}\ r := f;\ \ldots\end{aligned}$$

that it denotes a function such that there exist arbitrarily large integer arguments for which the function result is non-zero. The formalisation of the informal argument leads us to define, for worlds w containing a function location $\rho(r) \in \mathsf{Loc}_{\mathsf{int}\Rightarrow\mathsf{int}}$ and $h \in [\![\mathsf{int} \Rightarrow \mathsf{int}]\!]_w$:

$$\begin{aligned}h \in P_w \iff\ & \forall w' \geq w\ \forall s \in S_{w'}\ \forall n \in [\![\mathsf{int}]\!]_{w'}.\ s.(\rho(r)) \in P_{w'}\ \wedge\ h_{w'}(s,n)\downarrow\\ \Longrightarrow\quad & \exists m \in [\![\mathsf{int}]\!]_{w'}.\ m \geq n\\ \wedge\ & \forall w'' \geq w'\ \forall s'' \in S_{w''}\ \forall m' \in [\![\mathsf{int}]\!]_{w''}.\ h_{w'}(s,m) = \langle w'', \langle s'', m'\rangle\rangle \Longrightarrow m' \neq 0\end{aligned}$$

Note the existential quantification in the right-hand side, which has the consequence that this predicate is not closed under taking least upper bounds. Indeed, a simple counter-example is given by the chain $(h_i)_i$ in P_w, where $h_i(s,n) \stackrel{def}{=} \langle w, \langle s, 0\rangle\rangle$ if $n < i$ and undefined otherwise. The least upper bound h of this chain is everywhere defined and 0, thus cannot be an element of P_w.

Further note that this observation only means that the techniques of Chapter 2 are not applicable. Example (1.2) shows the stronger property that there exists no predicate P satisfying the above equivalence.

9.4 Remarks

Our semantic account of objects in this chapter closely follows the one considered in Kamin and Reddy's (1994) article, termed fixed point semantics. Classes and inheritance, as introduced in Section 3.4, could be explained directly, as proposed in (Kamin and Reddy 1994), too. However, in *loc. cit.* this interpretation was developed in an untyped setting, thus side-stepping two of the central issues we have had to deal with in our semantics: modelling types in the presence of *dynamic allocation of heap storage* and *subtyping*.

The interpretation of objects and classes sketched above also highlights another point which we believe deserves some discussion. At the end of (Kamin and Reddy 1994) where the closure semantics is compared to their alternative self-application model of objects, the advantage of the former with respect to typed languages is correctly anticipated. They remarked that "[because] self-application uses universal reflexive domains, the fixed point approach is better suited for typed languages." Self-application here refers to the self-parameter of methods (which is then bound to the host object at method invocation time, as explained in Section 3.2) rather than the store. It is well-known that, in a typed language, this contra-variant occurrence of the self-type in the type signature of methods blocks desirable subtypings, again we refer to the beginning of Chapter 3 for a more comprehensive discussion.

We want to point out that the above quotation from (Kamin and Reddy 1994) may need some clarification: Of course there is self-application also in the *imperative* fixed-point model, namely of the store parameter. This is due to the higher-order store; as explained in Section 8.4, the domains used to interpret the language of Section 8.3 are also reflexive domains. That this use of reflexive domains seems unavoidable is witnessed by programs using recursion through the store, such as the factorial example of Section 9.3. However, the store parameter remains implicit; in particular, it does not appear in the source-level type of the methods of an object and thus does not interfere with subtyping.

Chapter 10

Discussion

This concluding chapter is organised as follows. After a comparison to related work, we also discuss the relation of the typed model to the semantic model of Abadi-Leino logic in Part II. Then, in the remainder of this chapter, some initial ideas for local reasoning about higher-order store are developed. In particular, the intrinsic model is refined to include a notion of *local computation* which is shown to hold of all elements definable in the language.

We conclude with a summary of our results and identify a number of directions for future work.

10.1 Comparison to Related Work

Apart from Levy's work (Levy 2002; Levy 2004) which we built upon here, we are aware of only few other typed semantic models of higher-order store in the literature. The models in (Abramsky, Honda, and McCusker 1998; Laird 2003) use games semantics and are not location-based, i.e., the store is modelled only indirectly via possible program behaviours. Neither of these articles considers subtyping. Jeffrey and Rathke (2002) provide a model of the object calculus in terms of interaction traces, very much in the spirit of games semantics, which can be used to prove contextual equivalences.

Ahmed, Appel and co-workers (Ahmed, Appel, and Virga 2002; Appel and McAllester 2001; Ahmed, Appel, and Virga 2003; Ahmed, Fluet, and Morrisett 2005) follow a different approach by constructing models with a rather operational flavour: The semantics of types is obtained by approximating absence of type errors in a reduction semantics. While they can model rich type systems in this way, their models also suffer from the fact that allocated locations are globally accessible, i.e., encapsulation is not modelled.

Other related work we have already mentioned are the recent models of languages with pointers, but no higher-order store: Reddy and Yang (2004) and Benton and Leperchey (2005) provide semantic accounts of encapsulation and data abstraction for programs with dynamically allocated heap memory where pointers may be leaked. In contrast, Banerjee and Naumann (2002) solve the problem of modelling representation indepen-

dence by ruling out potential access to private memory from the outside, by imposing a confinement condition on programs.

There is a vast body of work on interpreting objects in procedural languages. Bruce, Cardelli and Pierce (1999) provide a fairly comprehensive overview and comparison of the more successful encodings of typed objects. Most of the encodings in the literature consider functional objects only. Exceptions are the denotational analyses of objects and inheritance by Kamin and Reddy (1994), and Cook and Palsberg (1994). However, they use untyped models, side-stepping the issues of modelling types and subtyping in the presence of updates and dynamic allocation. Many others apply similar ideas in a purely syntactic setting (Bono, Patel, Shmatikov, and Mitchell 1999; Thorup and Tofte 1994; Abadi, Cardelli, and Viswanathan 1996; Boudol 2004). This suffices for proofs of type soundness, but more expressive specifications are not usually discussed, and indeed logics of languages with higher-order store are hard to justify in such a setting. See our discussion in Section 7.1.1 of the soundness proof of (Abadi and Leino 2004).

The proof principles for recursion through the store and call-backs applied in Section 9.3 are direct adaptations of those presented by Reus and Streicher (2004): They developed them in the context of the untyped model of the object calculus, given in Section 3.3.3. The call-back example in particular shows that there is a trade-off in writing down predicates with respect to the typed and untyped models: On the one hand, assuming a typed model often allows for more concise specifications. On the other, the possible worlds structure requires one to define families of predicates instead of a single predicate.

Honda, Berger and Yoshida (Honda, Yoshida, and Berger 2005; Berger, Honda, and Yoshida 2005) present program logics for higher-order procedures and general references. Soundness is proved with respect to a term model. It is not clear to us to which extent their proof system can prove properties of higher-order store: They deal with *total* correctness assertions and thus can use well-founded induction on a termination order in their examples of recursion-through-the-store. But relying on total correctness also entails that even very simple *non-terminating* programs cannot be proved as such. Interestingly, the logic of (Honda, Yoshida, and Berger 2005) is based on naming (intermediate) results. This brings it close to Abadi and Leino's logic where objects are kept in the store and can therefore be referred to by a unique location name.

10.1.1 Comparing the Typed and Untyped Models of Objects

We have considered untyped and typed models of the object calculus in Part II and Part III, respectively. It is natural to ask what their respective advantages and disadvantages are. We attempt to answer some questions here.

Firstly, reasoning directly about denotations of programs is somewhat simpler in the untyped model since one usually has to define a single predicate over the domain. In contrast, one must consider families of predicates, indexed by possible worlds, in the typed model. However, reasoning in the untyped model sometimes necessitates to explicitly state that certain fields and methods are defined in an object, which is automatically

guaranteed in a typed semantics.

Secondly, modelling the aspect of dynamic allocation adequately in the untyped model turns out to be problematic. In particular, establishing the semantics of store specifications in Chapter 5 is awkward; the construction of the domain of choice functions appears to be an ad-hoc solution. As long as one is only interested in type soundness it seems much more elegant to rely on the "built-in" indexing of the typed model. Nevertheless, the extrinsic per model of Section 8.8.1 provides a link between typed and untyped model, using the intrinsic semantics to keep track of allocated locations.

Finally, we did not succeed in designing a program logic, similar to the one of Abadi and Leino, for the higher-order language of Section 8.3. The problem is the following: In general, specifications will contain free variables, i.e., depend on values. For the object calculus we have *syntax* for every value, including the run-time values. Thus, using substitution instances we can in fact restrict attention to closed specifications. In a language with higher-order functions this is not possible, so one really needs to find an appropriate model of this dependency. At the time of writing it is not clear to us how to refine the typed model in this way. In particular, naively extending the possible worlds to specifications that contain variables (analogous to the store specifications from Chapter 5) does not seem to work.

10.2 Outlook: Towards Local Reasoning for Higher-Order Store

We sketch an improvement to the intrinsic semantic model of Chapter 8 next. Informally, the property we establish states that a computation is already completely determined by the accessible part of the store. Formally this amounts to showing a *frame property* for general references. The recent work on *local reasoning* (Reynolds 2002a; O'Hearn, Reynolds, and Yang 2001) has identified frame properties as a semantic justification for frame rules.

10.2.1 Partial Stores

For local reasoning the notion of *partial stores* is a useful concept (Reynolds 2002a; O'Hearn, Reynolds, and Yang 2001). We define partial stores next, in the obvious way.

Definition 10.2.1 (Partial Store)**.** *For $w' \geq w$ let the cpo $S_w^{w'}$ of* partial stores *be*

$$S_w^{w'} \stackrel{def}{=} \{\!|\, l_A : [\![A]\!]_{w'} \,|\!\}_{l_A \in w}$$

In particular, $S_w^w = S_w$, and if $w = w_1 \uplus w_2$ then $s_1 \in S_{w_1}^w$ and $s_2 \in S_{w_2}^w$ implies $s_1 + s_2 \in S_w$.

Definition 10.2.2 (Store Embeddings)**.** *By a slight abuse of notation (suppressing w), define* store embeddings $\iota_{w_1}^{w_2} : S_w^{w_1} \to S_w^{w_2}$ *between partial stores by*

$$\iota_{w_1}^{w_2}(s) \stackrel{def}{=} \{\!|\, l_A = [\![A]\!]_{w_1}^{w_2}(s.l_A) \,|\!\}_{l_A \in w}$$

If there is s_2 such that $s = \iota_{w_1}^{w}(s_1) + s_2$ then we say s_1 *embeds in* s.

Relations on Partial Stores

In the context of working with partial stores, the relations of interest are *partial store relations*: Families $\mathcal{P} = (P_w^{w'})_{w' \geq w}$ of (admissible and downwards closed) predicates over partial stores, where $P_w^{w'} \subseteq S_w^{w'}$. In the obvious way call such a partial store relation *Kripke monotonic* iff

$$\forall w_2 \geq w_1 \geq w \; \forall s \in S_w^{w_1}.\; s \in P_w^{w_1} \quad \Longrightarrow \quad \iota_{w_1}^{w_2}(s) \in P_w^{w_2}$$

We observe that Kripke monotonicity is different from the notion of stability that was used earlier in Section 9.3 for the proof of object introduction: For instance, $\mathcal{P} = (P_w^{w'})$ where $s \in P_w^{w'}$ iff $s.l = 0$ for all $l \in w \cap \mathsf{Loc}_{\mathsf{int}}$ is monotonic but not stable. Conversely, the family $\mathcal{Q} = (Q_w^{w'})$ with $Q_w^{w'} = S_w$ if $w = w'$, and $Q_w^{w'} = \emptyset$ otherwise, is stable but not monotonic.

In essence, Kripke monotonicity means properties are preserved across embeddings of stores, while stability guarantees properties are preserved when adjoining a store to another (partial) store.

10.2.2 The Frame Property

Separation and a Frame Rule

Recall from Section 9.3 that by $\mathcal{A}dm(X)$ we denote the set of families of admissible predicates on $S \times X$, for $X \in [\mathcal{W}, \mathbf{pCpo}]$. We define a semantic (non-symmetric!) separation conjunction between admissible predicates and Kripke monotonic partial store relations next.

Definition 10.2.3 (Separation Conjunction)**.** *Let $\mathcal{P} = (P_w)_w \in \mathcal{A}dm(X)$ for some $X \in [\mathcal{W}, \mathbf{pCpo}]$. Let $\mathcal{Q} = (Q_w^{w'})_{w' \geq w}$ be a Kripke monotonic partial store relation. Define a partial store relation $\mathcal{P} \circledast \mathcal{Q} = (R_w)_w$ by*

$$\begin{aligned} \langle s, x \rangle \in R_w \quad \overset{def}{\Longleftrightarrow} \quad & \exists w_1, w_2 \in \mathcal{W} \; \exists s_1 \in S_{w_1} \; \exists s_2 \in S_{w_2}^{w'} \; \exists x_1 \in X_{w_1}. \\ & w = w_1 \uplus w_2 \;\wedge\; s = \iota_{w_1}^{w'}(s_1) + s_2 \;\wedge\; x = X_{w_1}^{w}(x_1) \\ & \wedge\; \langle s_1, x_1 \rangle \in P_{w_1} \;\wedge\; s_2 \in Q_{w_2}^{w'} \end{aligned}$$

That is, s splits into a (complete) store s_1 embedded in s and a partial store s_2 that satisfy $\mathcal{P}$ and $\mathcal{Q}$, respectively. A consequence of this asymmetric definition is that no value stored in the s_1-part refers to s_2; moreover, x may only refer to locations in s_1. Conversely however, the contents of s_2 may well depend on locations of s_1.

For example, $\mathcal{Q} = (Q_w^{w'})$ could refer to the "missing" part of the store in a hypothetic way,

$$s_2 \in Q_w^{w'} \quad \overset{def}{\Longleftrightarrow} \quad \forall s_1 \in S_{w_1}.\; (s_1.l = 5 \;\Longrightarrow\; \ldots)$$

where $w' \overset{def}{=} w \uplus w_1$ for some $w_1 \in \mathcal{W}$ and $s'.l \in w_1 \cap \mathsf{Loc}_{\mathsf{int}}$. This is reminiscent of the resource interpretation of the *magic wand* connective $-\!\!*$ of logic BI (Pym, O'Hearn, and Yang 2004).

Next suppose $h \in [\![A \Rightarrow B]\!]_\emptyset = \prod_w S_w \times [\![A]\!]_w \to \sum_{w' \geq w} (S_{w'} \times [\![B]\!]_{w'})$. We use the notation for Hoare triples for families of admissible predicates, introduced on page 163. Our goal is to prove the soundness of *frame rules* such as

$$\frac{\{\mathcal{P}\}h\{\mathcal{Q}\}}{\{\mathcal{P} \circledcirc \mathcal{R}\}h\{\mathcal{Q} \circledcirc \mathcal{R}\}} \ \mathcal{R} \text{ Kripke monotonic} \tag{10.1}$$

where $\mathcal{P} \in \mathcal{A}dm([\![A]\!])$, $\mathcal{Q} \in \mathcal{A}dm([\![B]\!])$ and $\mathcal{R}$ is a Kripke monotonic partial store relation.

Thus (10.1) expresses that (separated) invariants $\mathcal{R}$ may be conjoined to any valid Hoare triple. Semantically, soundness of this reasoning relies on the intuitive valid property that computations act "locally", their result completely determined by the accessible part of the store. This is formalised as a frame property next.

Frame Property

We prove a *frame property* (Reynolds 2002a; O'Hearn, Reynolds, and Yang 2001) for the programs of the language of Section 8.3 next. The following definition introduces a notion of programs operating locally on the store, parameterised by families of relations P and Q. To avoid further cluttering of the already heavy notation, we write ww_1 for a disjoint union of worlds $w \uplus w_1$.

Definition 10.2.4 (P-Q-local Computations)**.** *Suppose P and Q are families of predicates $P^{St}_w \subseteq S_w$ and $P^A_w \subseteq [\![A]\!]_w$, and $Q^{St}_w \subseteq S_w$ and $Q^A_w \subseteq [\![A]\!]_w$, for all $w \in \mathcal{W}$ and $A \in Type$. Further let $w \geq w_0$. We say $h \in [\![A \Rightarrow B]\!]_{w_0}$ is w-P-Q-*local *iff,*

$$\begin{array}{l}
\forall s \in S_w \ \forall a \in [\![A]\!]_w \,. \\
\forall w_1 \in \mathcal{W} \ \forall w_2 \geq w_1 \uplus w \ \forall s_1 \in S^{ww_1}_{w_1} \ \forall s_2 \in S_{w_2} \ \forall b_2 \in [\![B]\!]_{w_2} \,. \\
\quad (\iota^{ww_1}_w(s) + s_1) \in P^{St}_{ww_1} \ \wedge \ [\![A]\!]^{ww_1}_w(a) \in P^A_{ww_1} \ \wedge \\
\quad h_{ww_1}(\iota^{ww_1}_w(s) + s_1, [\![A]\!]^{ww_1}_w(a)) = \langle w_2, \langle s_2, b_2 \rangle \rangle \ \Longrightarrow \\
\qquad \exists w' \geq w, \ s' \in S_{w'} \ b \in [\![B]\!]_{w'} \,. \\
\qquad\quad h_w(s,a) = \langle w', \langle s', b \rangle \rangle \ \wedge \ w_2 = w' \uplus w_1 \ \wedge \\
\qquad\quad s_2 = \iota^{w_2}_{w'}(s') + s'_1 \ \wedge \ s_2 \in Q^{St}_{w_2} \ \wedge \ b_2 = [\![B]\!]^{w_2}_{w'}(b) \ \wedge \ b_2 \in Q^B_{w_2}
\end{array}$$

for some $s'_1 \sqsubseteq \iota^{w_2}_{ww_1}(s_1)$.

*We simply say h is P-Q-*local *if it is w'-P-Q-local for all $w' \geq w$, and P-*local *if it is P-P-local.*

This situation may be pictured as follows. Fix a partial store $s_1 \in S^{ww_1}_{w_1}$, and define the embeddings $\varepsilon_1(s) \stackrel{def}{=} \iota^{ww_1}_w(s) + s_1$ and $\varepsilon_2(s) \stackrel{def}{=} \iota^{w'w_1}_{w'}(s) + \iota^{w'w_1}_{ww_1}(s_1)$. The following diagram commutes (more precisely, the upper-left map is less or equal than the lower-right one):

$$\begin{array}{ccc}
S_{ww_1} \times [\![A]\!]_{ww_1} & \xleftarrow{\ \varepsilon_1 \times [\![A]\!]^{ww_1}_w\ } & S_w \times [\![A]\!]_w \\
\Big\downarrow{\scriptstyle h_{ww_1}} & & \Big\downarrow{\scriptstyle h_w} \\
\sum_{w_2 \geq ww_1} S_{w_2} \times [\![A]\!]_{w_2} & \xleftarrow[\ \sum_{w'} \varepsilon_2 \times [\![B]\!]^{w'w_1}_{w'}\ ]{} & \sum_{w' \geq w} S_{w'} \times [\![B]\!]_{w'}
\end{array}$$

Due to the typing assumptions, the partial store s_1 is not accessible by the computation. Definition 10.2.4 formalises the idea that in this situation (1) s_1 remains unchanged, and

Table 10.1 *Frame relation*

$$
\begin{array}{ll}
R^{\mathsf{bool}}_w & = [\![\mathsf{bool}]\!]_w \\
R^{\{\mathsf{m}_i:A_i\}_{i\in I}}_w & = \{r \in [\![\{\mathsf{m}_i : A_i\}_{i\in I}]\!]_w \mid r.\mathsf{m}_i \in R^{A_i}_w \quad \forall i \in I\} \\
R^{\mathsf{ref}\,A}_w & = [\![\mathsf{ref}\,A]\!]_w \\
R^{A\Rightarrow B}_w & = \{h \in [\![A \Rightarrow B]\!]_w \mid h \text{ is } R\text{-local}\} \\
R^{St}_w & = \{s \in S_w \mid s.l_A \in R^A_w \quad \forall l_A \in w\}
\end{array}
$$

(2) the result of the computation is the same irrespective of whether s_1 is or is not conjoined.

Table 10.1 defines a Kripke logical relation R with $R^A_w \subseteq [\![A]\!]_w$ that holds if "R-locality of stores" is preserved. Because of the higher-order store, this can only hold of programs when applied to stores that contain nothing but R-local values. Then, because of the negative occurrence of R in the clause for function types, existence of R needs to be established. The existence proof illustrates why we can only require $s'_1 \sqsubseteq \iota^{w_2}_{ww_1}(s_1)$ – rather than equality – in Definition 10.2.4. Fortunately this is still sufficient for the partial correctness assertions we are interested in.

For the purpose of proving well-definedness of the logical relation R, we consider a "Kripke" relational structure over the category

$$C \;=\; \textstyle\prod_{w\in\mathcal{W}} \mathbf{pCpo} \times \prod_{A\in Type}[\mathcal{W}, \mathbf{Cpo}] \bullet\!\!\to [\mathcal{W}, \mathbf{pCpo}]$$

as follows: Relations over an object $\langle D_S, D\rangle$ are families P of admissible downward closed subsets $P^{St}_w \subseteq D_{Sw}$ and $P^A_w \subseteq D_{Aw}$ such that $a \in P^A_w$ implies $D_{A,w\le w'}(a) \in P^A_{w'}$ for all $w' \geq w$, and where for $e : \langle D_S, D\rangle \to \langle E_S, E\rangle$, $e : P \subset Q$ holds iff

$$
\begin{array}{l}
\forall w \in \mathcal{W}\ \forall d \in D_{Aw}.\ d \in P^A_w \ \wedge\ e_{Aw}(d)\downarrow \ \implies\ e_{Aw}(d) \in Q^A_w \\
\forall w \in \mathcal{W}\ \forall s \in D_{Sw}.\ s \in P^{St}_w \ \wedge\ e_{Sw}(s)\downarrow \ \implies\ e_{Sw}(s) \in Q^{St}_w
\end{array}
$$

Table 10.2 defines a functional Φ such that the definition of the relation R reduces to finding a fixed point $R = \Phi(R, R)$. The next lemma shows that Φ defines an admissible action of the functor F of Section 8.4 (Table 8.3) for which the semantic domain $D = \langle (S_w)_{w\in\mathcal{W}}, ([\![A]\!])_{A\in Type}\rangle$ is the minimal invariant:

Lemma 10.2.5. *Let $I = \{\delta^n(\bot) \mid n \in \mathbb{N}\}$ be the set of projections $\delta^n(\bot)$. Let $e \in I$ and assume that $e : P' \subset P$ and $e : Q \subset Q'$. Then $F(e, e) : \Phi(P, Q) \subset \Phi(P', Q')$.*

Proof sketch. The interesting case is for the components of the form $A \equiv B{\Rightarrow}B'$. Thus if $h \in \Phi(P, Q)_{Aw_0}$ and $F(e, e)_{Aw_0}(h)\downarrow$ we must show $F(e, e)_{Aw_0}(h) \in \Phi(P', Q')_{Aw_0}$, i.e., $F(e, e)_{Aw_0}(h)$ is P'-Q'-local.

So let $w \geq w_0$, let $s \in S^{St}_w$ and $a \in [\![A]\!]_w$. Let $w_1 \in \mathcal{W}$, let $s_1 \in S^{ww_1}_{w_1}$ such that $(\iota^{ww_1}_w(s){+}s_1) \in {P'}^{St}_{ww_1}$ and $[\![A]\!]^{ww_1}_w(a) \in {P'}^A_{ww_1}$, and suppose there are $w_2 \geq w \uplus w_1$ and $s_2 \in S_{w_2}$ and $b_2 \in [\![B]\!]_{w_2}$ such that

$$F(e, e)_{Aw_0}(h)_{ww_1}(\iota^{w,w_1}_w(s){+}s_1, [\![A]\!]^{ww_1}_w(a)) \;=\; \langle w_2, \langle s_2, b_2\rangle\rangle$$

Table 10.2 *Functional of the relation*

$b \in \Phi(P,Q)^{\mathsf{bool}}_w$	$\overset{def}{\iff}$	$b \in [\![\mathsf{bool}]\!]_w$
$r \in \Phi(P,Q)^{\{\mathsf{m}_i:A_i\}_{i\in I}}_w$	$\overset{def}{\iff}$	$r \in [\![\{\mathsf{m}_i : A_i\}_{i\in I}]\!]_w \ \wedge\ r.\mathsf{m}_i \in Q^{A_i}_w \quad \forall i \in I$
$l \in \Phi(P,Q)^{\mathsf{ref}\ A}_w$	$\overset{def}{\iff}$	$l \in [\![\mathsf{ref}\ A]\!]_w$
$h \in \Phi(P,Q)^{A\Rightarrow B}_w$	$\overset{def}{\iff}$	$h \in [\![A \Rightarrow B]\!]_w \ \wedge\ h$ is P-Q-local
$s \in \Phi(P,Q)^{St}_w$	$\overset{def}{\iff}$	$s \in S_w \ \wedge\ s.l_A \in Q^A_w \quad \forall l_A \in w$

We must prove that there are $w' \geq w$; $s' \in S_{w'}$ and $b \in [\![B]\!]_{w'}$ such that

$$\begin{aligned} &h_w(s,a) = \langle w', \langle s', b\rangle\rangle \wedge \\ &w_2 = w' \uplus w_1 \wedge \\ &\exists s'_1 \sqsubseteq \iota^{w_2}_{ww_1}(s_1).\ s_2 = \iota^{w_2}_{w'}(s') + s'_1 \ \wedge\ s_2 \in {Q'}^{St}_{w_2} \wedge \\ &b_2 = [\![B]\!]^{w_2}_{w'}(b) \ \wedge\ b_2 \in {Q'}^{B}_{w_2} \end{aligned} \tag{10.2}$$

By definition of F, we know that $s_2 = e_{Sw_2}(\hat{s_2})$ and $b_2 = e_{Bw_2}(\hat{b_2})$ where

$$\langle w_2, \langle \hat{s_2}, \hat{b_2}\rangle\rangle = h_{w,w_1}(e_{Sww_1}(\iota^{ww_1}_w(s) + s_1),\ e_{Aww_1}([\![A]\!]^{ww_1}_w(a)))$$

First observe that the projections are below the identity and act pointwise on stores. Therefore we have $e_{Sww_1}(\iota^{ww_1}_w(s) + s_1) = \iota^{ww_1}_w(e_{Sw}(s)) + s''_1$ for some $s''_1 \sqsubseteq s_1$ in $S^{ww_1}_{w_1}$. By the assumption $e : P' \subset P$,

$$\iota^{ww_1}_w(e_{Sw}(s)) + s''_1 = e_{Sww_1}(\iota^{ww_1}_w(s) + s_1) \in P^{St}_{ww_1}$$

Similarly, $[\![A]\!]^{ww_1}_w(e_{Aw}(a)) = e_{Aww_1}([\![A]\!]^{ww_1}_w(a)) \in P^A_{ww_1}$, so the assumption $h \in \Phi(P,Q)_{w_0}$ entails that there exists $w' \geq w$, $\hat{s} \in S_{w'}$ and $\hat{b} \in [\![B]\!]_{w'}$ such that

$$\begin{aligned} &h_w(e_{Sw}(s), e_{Aw}(a)) = \langle w', \langle \hat{s}, \hat{b}\rangle\rangle \\ &w_2 = w' \uplus w_1 \wedge \\ &\hat{s_2} = \iota^{w_2}_{w'}(\hat{s}) + s'' \ \wedge\ \hat{s_2} \in Q^{St}_{w_2} \ \wedge \\ &\hat{b_2} = [\![B]\!]^{w_2}_{w'}(\hat{b}) \ \wedge\ \hat{b_2} \in Q^B_{w_2} \end{aligned}$$

for some $s'' \sqsubseteq \iota^{w_2}_{ww_1}(s''_1)$. By assumption $e : Q \subset Q'$ this entails $s_2 = e_{Sw_2}(\hat{s_2}) \in {Q'}^{St}_{w_2}$ and $b_2 = e_{Bw_2}(\hat{b_2}) \in {Q'}^{B}_{w_2}$. Furthermore,

$$b_2 = e_{Bw_2}(\hat{b_2}) = e_{Bw_2}([\![B]\!]^{w_2}_{w'}(\hat{b})) = [\![B]\!]^{w_2}_{w'}(e_{Bw'}(\hat{b}))$$

In particular, $e_{Bw'}(\hat{b})\downarrow$ and $b_2 = [\![B]\!]^{w_2}_{w'}(b)$ where $b \overset{def}{=} e_{Bw'}(\hat{b})$. Also

$$\begin{aligned} s_2 = e_{Sw_2}(\hat{s_2}) &= e_{Sw_2}(\iota^{w_2}_{w'}(\hat{s}) + s'') \\ &= \{\!|\, l = e_{Aw_2}([\![A]\!]^{w_2}_{w'}(\hat{s}.l)) \,|\!\}_{l_A \in w'} + s'_1 \\ &= \{\!|\, l = [\![A]\!]^{w_2}_{w'}(e_{Aw_2}(\hat{s}.l)) \,|\!\}_{l_A \in w'} + s'_1 \\ &= \iota^{w_2}_{w'}(e_{Sw'}(\hat{s})) + s'_1 \end{aligned}$$

for some $s_1' \sqsubseteq s''$, which implies $e_{Sw'}(\hat{s})\downarrow$. Let $s' \stackrel{def}{=} e_{Sw'}(\hat{s})$. It follows that $s_2 = \iota_{w'}^{w_2}(s') + s_1'$ where $s_1' \sqsubseteq \iota_{ww_1}^{w_2}(s_1)$ and since $F(e,e)_{Aw_0}(h)_w(s,a) = \langle w', \langle s', b\rangle\rangle$ by definition of F, we have shown the requirements (10.2). □

By the existence theorem, Theorem 2.5.3, we can conclude that $R \stackrel{def}{=} \mathit{fix}(\Phi)$ is well-defined. The relation R extends to contexts in the obvious way, by $\rho \in R_w^\Gamma$ if and only if $\rho(x) \in R_w^A$ for all $x{:}A$ in Γ. We would like to proceed by proving a logical relations lemma for R next. For the particular case of closed terms this would state precisely the frame condition we are after.

Desired Property 10.2.6 (Basic Lemma, Frame Property)**.** *Suppose* $\Gamma \rhd e : B$. *Assume* $w, w_1 \in \mathcal{W}$ *such that* $w \cap w_1 = \emptyset$. *For all* $\rho \in R_w^\Gamma$, *for all* $s \in S_w$ *and* $s_1 \in S_{w_1}^{ww_1}$ *such that* $(\iota_w^{ww_1}(s) + s_1) \in R_{ww_1}^{St}$, *if*

$$[\![\Gamma \rhd e : B]\!]_{ww1}\,([\![\Gamma]\!]_w^{ww_1}(\rho))(\iota_w^{ww_1}(s)+s_1) = \langle w_2, \langle s_2, b_2\rangle\rangle$$

for some $w_2 \geq w \uplus w_1$, $s_2 \in S_{w_2}$ *and* $b_2 \in [\![B]\!]_{w_2}$, *then there exist* $w' \geq w$, $s' \in S_{w'}$ *and* $b \in [\![B]\!]_{w'}$ *such that*

$$[\![\Gamma \rhd e : B]\!]_w\,\rho s = \langle w', \langle s', b\rangle\rangle \;\wedge\; w_2 = w' \uplus w_1 \;\wedge$$
$$s_2 = \iota_{w'}^{w_2}(s') + s_1' \;\wedge\; s_2 \in R_{w_2}^{St} \;\wedge\; b_2 = [\![B]\!]_{w'}^{w_2}(b) \;\wedge\; b_2 \in R_{w_2}^B$$

for some $s_1' \sqsubseteq \iota_{ww_1}^{w_2}(s_1)$.

A proof of this statement would rely on the following properties of R:

- Downward closure: $a \in R_w^A \;\wedge\; a' \sqsubseteq a \;\Longrightarrow\; a' \in R_w^A$
- Kripke monotonicity: $a \in R_w^A \;\wedge\; w' \geq w \;\Longrightarrow\; [\![A]\!]_w^{w'}(a) \in R_{w'}^A$
- Subtype monotonicity: $a \in R_w^A \;\wedge\; A \preceq B \;\Longrightarrow\; [\![A \preceq B]\!](a) \in R_w^B$

and proceed by induction on the derivation of $\Gamma \rhd e : B$.

Unfortunately there is a subtle problem lurking here. Implicit in our treatment of the choice of new locations we assume disjointness of the location sets w' and w_1. But this is not justified by our semantics. More precisely, after splitting off the inaccessible w_1-part of the store, the program is run on the remaining w-part, yielding a store over locations $w' \geq w$. The freshness condition for allocation of new storage in the semantics (see Table 8.6) thus applies only to w, in particular it fails to guarantee that new locations do not already appear in w_1. Concretely, this problem surfaces in a proof attempt of Statement 10.2.6 in the case of allocation.

In other words, an allocation mechanism with only "local knowledge" cannot take information about the inaccessible part of the store into account in order to determine the next free location. We expect a formally correct treatment to be achievable using Pitts and Shinwell's FM domains (Shinwell 2005). In this setting, there is a group action on the set of locations by the permutation group, which formalises α-renaming. This corresponds to the informal understanding that the actual name of a location is irrelevant, as long as

it is used consistently. In forthcoming work, we take a related approach and study the problem in a functor category where $\mathcal{W}$ contains all injections $w \to w'$, rather than just the embeddings (Reus and Schwinghammer 2006b). By taking renamings seriously we arrive at a suitable definition of frame properties for such a model.

However, modulo this problem concerning the choice of locations, we can already show soundness of the frame rule. In particular, the frame rule is sound for the sublanguage that does not use dynamic allocation.

Soundness of the Frame Rule

Consider the frame rule (10.1) again:

$$\frac{\{\mathcal{P}\}h\{\mathcal{Q}\}}{\{\mathcal{P} \circledast \mathcal{P}'\}h\{\mathcal{Q} \circledast \mathcal{P}'\}} \; \mathcal{B} \text{ and } \mathcal{P}' \text{ Kripke monotonic}$$

where $\mathcal{P}$ and $\mathcal{Q}$ have to be interpreted as families of admissible relations over the sub-cpos R^A of $[\![A]\!]$, rather than $[\![A]\!]$ itself.

To see soundness of the frame rule for programs of the higher-order language from Section 8.3, let $h \in R^{A \Rightarrow B}_{\emptyset}$. In particular, by the Frame Property (Lemma 10.2.6), h could be the denotation of a program of type $A \Rightarrow B$, i.e.,

$$\langle \emptyset, \langle s, h \rangle \rangle \stackrel{def}{=} [\![\triangleright \lambda x.\, e : A \Rightarrow B]\!]_{\emptyset}$$

Suppose $\{\mathcal{P}\}\, h\, \{\mathcal{Q}\}$ holds, and let $w \in \mathcal{W}$. Further suppose that $a \in R^A_w$ and $s \in R^{St}_w$ are such that $\langle s, a \rangle \in (P \circledast P')_w$. By definition this means there are $w_1, w_2 \in \mathcal{W}$ such that $w = w_1 \uplus w_2$, and there are $a_1 \in R^A_{w_1}$, $s_1 \in R^{St}_{w_1}$ and $s_2 \in S^w_{w_2}$ such that $s = \iota^w_{w_1}(s_1) + s_2$ and $a = [\![A]\!]^w_{w_1}(a_1)$, and moreover $\langle s_1, a_1 \rangle \in P_{w_1}$ and $s_2 \in {P'}^w_{w_2}$.

Suppose $h_w s$ is defined, i.e., $h_w(s, a) = \langle w_3, \langle s_3, b \rangle \rangle$ for some $w_3 \geq w$, $s_3 \in S_{w_3}$ and $b \in [\![B]\!]_{w_3}$. By the definition of the logical relation h is R-local, so there exist $w' \geq w_1$, $s' \in S_{w_1}$ and $b' \in [\![B]\!]_{w'}$ such that

- $h_{w_1}(s_1, a_1) = \langle w', \langle s', b' \rangle \rangle$ and $w_3 = w' \uplus w_2$;
- $s_3 = \iota^{w_3}_{w'}(s') + s'_2$ for some $s'_2 \sqsubseteq s_2$; and
- $b = [\![B]\!]^{w_3}_{w'}(b')$.

From assumption $\{\mathcal{P}\}\, h\, \{\mathcal{Q}\}$ we obtain $\langle s', b' \rangle \in Q_{w'}$. By Kripke monotonicity of $\mathcal{P}'$, $\iota^{w_3}_{w'}(s_2) \in {P'}^{w_3}_{w_2}$, and therefore also $s'_2 \in {P'}^{w_3}_{w_2}$ by downward closure. Thus, $\langle s_3, b \rangle \in (P \circledast P')_{w_3}$.

Remark 10.2.7. *We were careful in the asymmetric definition of the separation conjunction to ensure that the argument does not accidentally provide access into the part of the store for which the separated relation $\mathcal{P}'$ holds. Otherwise simple counterexamples to the frame rule can be constructed, based on programs like the following*

$$\triangleright \lambda x.\, x{:=}(\mathsf{deref}\ x) + 1 : \mathsf{ref\ int} \Rightarrow \mathbf{1}$$

which may invalidate, e.g., the invariant $s.l = 5$, when applied to reference l.

10.3 Summary and Conclusions

Programming languages that use general references and higher-order store are common in practice. Standard ML and object-based languages are examples. Perhaps less obviously so, advanced features of class-based languages such as class-loading and inner classes also lead to similar semantic domains by blurring the distinction between class tables and data store.

In this thesis we made a number of contributions to the understanding of semantics and logics for higher-order storage, focussing on Abadi and Cardelli's imperative object calculus.

Part II demonstrated that the techniques presented previously in (Reus and Streicher 2002; Reus and Streicher 2004) can be extended to the full proof calculus of Abadi and Leino (2004), although not as directly as we had originally anticipated. We showed soundness of the logic with respect to a denotational semantics and discussed extensions, in particular the introduction of recursive specifications. The main difference to (Reus and Streicher 2004) is in the shift in perspecitve from individual objects to stores, necessitated by the compositionality of the logic. In particular, recursively defined object specifications have been replaced by recursively defined store specifications.

Part III was devoted to an intrinsically typed model of the object calculus, where subtyping is interpreted by coercions. We proceeded indirectly, by extending the known possible worlds model of Levy (2002, 2004) with subtyping, and then interpreting objects via a syntactic translation. The main technical contribution of Chapter 8 is the coherence result (Theorem 8.7.1). Its proof relied on the Basic Lemma of a logical relation (Lemma 8.6.7) and the properties of certain retractions ("bracketing maps"). To obtain a relation with the required properties we needed to show its well-definedness (Theorem 8.6.4). Yet another model, based on partial equivalence relations, was obtained as a corollary to the Bracketing theorem (Theorem 8.6.8).

The actual interpretation of the object calculus turned out to be comparatively simple, closely following the ideas of (Kamin and Reddy 1994). Chapter 9 also contained several examples proving non-trivial properties of objects, in some cases using recursion through the store. In this final chapter we have developed some initial ideas for local reasoning in the presence of higher-order store, which we plan to investigate more properly in future work.

Finally we mention a number of open problems and directions for future work.

Logics for Objects. In Section 7.2 we have already mentioned several potential extensions and variations of Abadi and Leino's logic that seem important. Of those, reasoning about invariants and access control will likely need a model that validates aspects of encapsulation.

In a different direction, one idea is to investigate a logic where store specifications are not necessarily preserved. For instance, different properties could hold of the contents of a field at various times during a computation. This bears some resemblance to the reuse of memory in low-level languages, for example in typed assembly language (Morrisett,

Ahmed, and Fluet 2005).

Logics for Classes. The reduction of class-based languages to object-based languages outlined in Section 3.4 means that it is already possible to talk about classes in Abadi and Leino's logic.

Our plan is to more fully investigate how the logic can be used for reasoning about dynamic class loading and inner classes. Considering Abadi-Leino logic as a logic for class-based languages may also identify a number of further desirable extensions.

Logics for Higher-order Functions. Designing calculi for reasoning about higher-order functions with general references seems important, also from a practical point of view. To this end it may be worthwhile to explain the work of (Honda et al. 2005) in terms of a model more "semantic" than their term model.

Local Reasoning. Above we have made only a very preliminary first step, showing that procedures operate "locally" on the accessible store (i.e., a frame condition). We have also identified some classes of predicates (*stable* and *Kripke monotonic* families) that not only allowed the proofs to go through but also appear to express natural properties. However, a full investigation of their closure properties and possible proof rules remains to be done.

The fact that the Kripke worlds keep track of the possible (dangling) references between disjoint parts of the store lets us define the separation connective in the first place. In particular, *safety monotonicity* which states that computation with respect to larger stores does not introduce "memory faults", is an immediate consequence of typing in the intrinsic model. In contrast, in an untyped model one would have to *require* such a safety property.

One might also want to try and define a *local function space* constructor, in the category $[\mathcal{W}, \mathbf{pCpo}]$, rather than defining locality on top of the typed model.

Modelling encapsulation and representation independence, which go beyond the locality considered in this chapter, is likely to be a challenging problem for languages with higher-order store.

Subtyping Recursive Types in the Presence of Higher-order Store. In a different direction, we can extend the language with a more expressive type system: Recursive types feature prominently in the work on semantics of *functional* objects. In (Levy 2004) it is suggested that the construction of the intrinsic model also works for a variant of recursive types. We have not considered the combination with subtyping yet, but do not expect any difficulties.

Coherence. While the individual facts are much more intricate to prove than for the functional language considered in (Reynolds 2002b), the overall structure of the coherence proof in Chapter 8 is almost identical to *loc. cit.* This suggests it could be interesting to work out the general conditions needed for the construction, for example, using the setting of (Mitchell and Scedrov 1993).

Bibliography

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer.

Abadi, M., L. Cardelli, and R. Viswanathan (1996). An interpretation of objects and object types. In *Conference record of the 23rd Symposium on Principles of Programming Languages*, pp. 396–409. ACM Press.

Abadi, M. and K. R. M. Leino (1997). A logic of object-oriented programs. In M. Bidoit and M. Dauchet (Eds.), *Proceedings of Theory and Practice of Software Development*, Volume 1214 of *Lecture Notes in Computer Science*, pp. 682–696. Springer.

Abadi, M. and K. R. M. Leino (2004). A logic of object-oriented programs. In N. Dershowitz (Ed.), *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, Lecture Notes in Computer Science, pp. 11–41. Springer.

Abadi, M. and G. D. Plotkin (1990). A per model of polymorphism and recursive types. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pp. 355–365. IEEE Computer Society Press.

Abelson, H., R. K. Dybvig, C. T. Haynes, R. Rozas, N. I. Adams IV, D. P. Friedman, K. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, B. Brooks, C. Hanson, K. M. Pitman, and M. Wand (1998). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*(1), 7–105.

Abramsky, S., K. Honda, and G. McCusker (1998). A fully abstract game semantics for general references. In *Proceedings 13th Annual IEEE Symposium on Logic in Computer Science*, pp. 334–344. IEEE Computer Society Press.

Abramsky, S. and A. Jung (1994). Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Volume 3, pp. 1–168. Clarendon Press.

Abramsky, S. and G. McCusker (1998). Game semantics. In H. Schwichtenberg and U. Berger (Eds.), *Logic and Computation. Proceedings of the 1997 Marktoberdorf Summer School*. Springer.

Aceto, L., H. Hüttel, A. Ingólfsdóttir, and J. Kleist (2000). Relating semantic models for the object calculus. In C. Palamidessi and J. Parrow (Eds.), *Electronic Notes in Theoretical Computer Science*, Volume 7.

Ahmed, A., M. Fluet, and G. Morrisett (2005). A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM Press.

Ahmed, A. J., A. W. Appel, and R. Virga (2002). A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of 17th Annual IEEE Symposium Logic in Computer Science*, pp. 75–86. IEEE Computer Society Press.

Ahmed, A. J., A. W. Appel, and R. Virga (2003). An indexed model of impredicative polymorphism and mutable references. Princeton University.

Amadio, R. M. (1991). Recursion over realizability structures. *Information and Computation 91*(1), 55–86.

Amadio, R. M. and L. Cardelli (1993). Subtyping recursive types. *ACM Transactions on Programming Languages and Systems 15*(4), 575–631.

Appel, A. W. and D. McAllester (2001). An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems 23*(5), 657–683.

Apt, K. R. (1981). Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems 3*(4), 431–483.

Banerjee, A. and D. A. Naumann (2002). Representation independence, confinement and access control. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp. 166–177. IEEE Computer Society Press.

Barendregt, H. P. (1992). Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Volume 2, Chapter 2, pp. 117–309. Oxford University Press.

Benton, N. and B. Leperchey (2005). Relational reasoning in a nominal semantics for storage. In *To appear in Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications (TLCA '05)*, Lecture Notes in Computer Science. Springer.

Berger, M., K. Honda, and N. Yoshida (2005). A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM Press.

Birkedal, L. and R. W. Harper (1999). Constructing interpretations of recursive types in an operational setting. *Information and Computation 155*, 3–63.

Birkedal, L., N. Torp-Smith, and H. Yang (2005). Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press.

Bono, V., A. J. Patel, V. Shmatikov, and J. C. Mitchell (1999). A core calculus of classes and objects. In *15th Conference on the Mathematical Foundations of Programming Semantics*, Volume 20 of *Electronic Notes in Computer Science*.

Boudol, G. (2004). The recursive record semantics of objects revisited. *Journal of Functional Programming 14*(3), 263–315.

Bracha, G., M. Odersky, D. Stoutamire, and P. Wadler (1998). Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN Notices 33*(10), 183–200.

Breazu-Tannen, V., T. Coquand, G. Gunter, and A. Scedrov (1991). Inheritance as implicit coercion. *Information and Computation 93*(1), 172–221.

Bruce, K. B. (1994). A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming 4*(2), 127–206.

Bruce, K. B. (2002). *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press.

Bruce, K. B., L. Cardelli, and B. C. Pierce (1999). Comparing object encodings. *Information and Computation 155*(1/2), 108–133.

Cardone, F. (1989). Relational semantics for recursive types and bounded quantification. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca (Eds.), *16th International Colloquium Automata, Languages and Programming*, Volume 372 of *Lecture Notes in Computer Science*, pp. 164–178. Springer.

Castagna, G. (1997). *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser.

Clarke, E. M. (1979). Programming language constructs for which it it impossible to obtain good Hoare axiom systems. *Journal of the ACM 26*(1), 129–147.

Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing 7*(1), 70–90.

Cook, W. and J. Palsberg (1994). A denotational semantics of inheritance and its correctness. *Information and Computation 114*(2), 329–350.

Cook, W. R. (1989). *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University.

Cousot, P. (1990). Methods and logics for proving programs. In J. van Leeuwen (Ed.), *Formal Models and Semantics*, Volume B of *Handbook of Theoretical Computer Science*, Chapter 15, pp. 843–993. Elsevier.

Curien, P.-L. and G. Ghelli (1994). Coherence of subsumption, minimum subtyping and type-checking in $F_{\leq}$. In C. A. Gunter and J. C. Mitchell (Eds.), *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Foundations of Computing Series, pp. 247–292. MIT Press.

Davey, B. A. and H. A. Priestley (2002). *Introduction to Lattices and Order* (Second ed.). Cambridge University Press.

de Boer, F. S. (1999). A WP-calculus for OO. In W. Thomas (Ed.), *Foundations of Software Science and Computation Structures*, Volume 1578 of *Lecture Notes in Computer Science*, pp. 135–149. Springer.

de Boer, F. S. and C. Pierik (2004). How to cook a complete hoare logic for your pet oo language. In F. S. de Boer, M. M. Bonsangue, and S. Graf (Eds.), *Proceedings Formal Methods for Components and Objects: Second International Symposium (FMCO '03)*, Number 3188 in Lecture Notes in Computer Science, pp. 111–133. Springer.

Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico (1995). An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation 8*(4), 357–397.

Erkök, L. and J. Launchbury (2000). Recursive monadic bindings. *ACM SIGPLAN Notices 35*(9), 174–185.

Fecher, H. (1999). Denotational semantics of untyped object-based programming languages. Master's thesis, Technische Universität Darmstadt.

Fiore, M., A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark (1996). Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of the European Association for Theoretical Computer Science 59*, 227–256.

Fiore, M. P. (1996). *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press.

Floyd, R. W. (1967). Assigning meanings to programs. In J. T. Schwartz (Ed.), *Proceedings of Mathematical Aspects of Computer Science*, Volume 19 of *Proceedings of Symposia in Applied Mathematics*, pp. 19–32. American Mathematical Society.

Freyd, P., G. Rosolini, P. Mulry, and D. Scott (1992). Extensional PERs. *Information and Computation 98*(2), 211–227.

Freyd, P. J. (1991). Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini (Eds.), *Proceedings of 1990 Como Category Theory Conference*, Volume 1488 of *Lecture Notes in Mathematics*, pp. 95–104. Springer.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Ghica, D. R. (2001). A regular-language model for hoare-style correctness statements. In M. Leuschel, A. Podelski, C. Ramakrishnan, and U. Ultes-Nitsche (Eds.), *Proceedings of the Second International Workshop on Verification and Computational Logic (VCL '01)*, pp. 25–42.

Glimming, J. (2005). *Dialgebraic Semantics of Typed Object Calculi*. Licentiate thesis, Stockholm University.

Glimming, J. and N. Ghani (2004). Difunctorial semantics of object calculus. In *Proceedings WOOD '04: Workshop on Object-Oriented Developments*, Electronic Notes in Theoretical Computer Science. Elsevier. To appear.

Gordon, A. D., P. D. Hankin, and S. B. Lassen (1997). Compilation and equivalence of imperative objects. In *Proceedings of FST+TCS'97*, Volume 1346 of *Lecture Notes in Computer Science*, pp. 74–87.

Goubault-Larrecq, J., S. Lasota, and D. Nowak (2002). Logical relations for monadic types. In *Proc. 16th Int. Workshop Computer Science Logic (CSL'02)*, Volume 2471 of *Lecture Notes in Computer Science*, pp. 553–568. Springer.

Halpern, J. Y. (1984). A good Hoare axiom system for an Algol-like language. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 262–271. ACM Press.

Hensel, U., M. Huisman, B. Jacobs, and H. Tews (1998). Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin (Ed.), *Programming Languages and Systems—ESOP'98, 7th European Symposium on Programming*, Volume 1381 of *Lecture Notes in Computer Science*, pp. 105–121. Springer.

Hoare, C. A. R. (1969). An Axiomatic Basis of Computer Programming. *Communications of the ACM 12*, 576–580.

Hofmann, M. and B. Pierce (1996). Positive subtyping. *Information and Computation 126*(1), 11–33.

Honda, K. (2004). From process logic to program logic. *ACM SIGPLAN Notices 39*(9), 163–174.

Honda, K., N. Yoshida, and M. Berger (2005). An observationally complete program logic for imperative higher-order functions. In *Proceedings of 20th Annual IEEE Symposium Logic in Computer Science (LiCS'05)*. IEEE Computer Society Press. To appear.

Honsell, F., A. Pravato, and S. R. D. Rocca (1998). Structured Operational Semantics of a fragment of the language Scheme. *Journal of Functional Programming 8*(4), 335–365.

Igarashi, A. and B. C. Pierce (2000). On inner classes. In *Proceedings of the European Conference on Object-Oriented Programming*, Volume 1850 of *Lecture Notes in Computer Science*, pp. 129–153. Springer.

Igarashi, A., B. C. Pierce, and P. Wadler (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*(3), 396–450.

Ishtiaq, S. S. and P. W. O'Hearn (2001). BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices 36*(3), 14–26.

Jacobs, B. and E. Poll (2001). A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE'2001)*, Volume 2029 of *Lecture Notes in Computer Science*, pp. 284–299. Springer.

Jacobs, B. and E. Poll (2003). Coalgebras and monads in the semantics of Java. *Theoretical Computer Science 291*(3), 329–349.

Jacobs, B. and J. Rutten (1997). A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science 62*, 222–259.

Jacobs, B. P. F. (1996). Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H. J. Schek (Eds.), *Object-Orientation with Parallelism and Persistence*, pp. 83–103. Kluwer Academic Publishers.

Jeffrey, A. and J. Rathke (2002). A fully abstract may testing semantics for concurrent objects. In *Proceedings* 17^{th} *Annual Symposium on Logic in Computer Science*, pp. 101–112. IEEE Computer Society Press.

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

Kernighan, B. and D. Ritchie (1988). *The C Programming Language* (Second ed.). Prentice-Hall.

Kleymann, T. (1999). Hoare logic and auxiliary variables. *Formal Aspects of Computing 11*(5), 541–566.

Laird, J. (2003). A categorical semantics of higher-order store. In R. Blute and P. Selinger (Eds.), *Proceedings of the 9th Conference on Category Theory and Computer Science, CTCS '02*, Volume 69 of *Electronic notes in Theoretical Computer Science*, pp. 1–18. Elsevier.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*(3), 872–923.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal 6*(4), 308–320.

Leino, K. R. M. (1998). Recursive object types in a logic of object-oriented programs. In C. Hankin (Ed.), *7th European Symposium on Programming*, Volume 1381 of *Lecture Notes in Computer Science*, pp. 170–184. Springer.

Levy, P. B. (2002). Possible world semantics for general storage in call-by-value. In J. Bradfield (Ed.), *16th Workshop on Computer Science Logic (CSL'02)*, Volume 2471 of *Lecture Notes in Computer Science*. Springer.

Levy, P. B. (2004). *Call-By-Push-Value. A Functional/Imperative Synthesis*, Volume 2 of *Semantic Structures in Computation*. Kluwer.

Longley, J. (1995). *Realizability toposes and language semantics*. Ph. D. thesis, University of Edinburgh.

Longo, G. and E. Moggi (1991). Constructive natural deduction and its 'ω-set' interpretation. *Mathematical Structures in Computer Science 1*(2), 215–254.

MacQueen, D. B., G. D. Plotkin, and R. Sethi (1986). An ideal model for recursive polymorphic types. *Information and Control 71*(1–2), 95–130.

Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From operational semantics to domain theory. *Information and Computation 128*(1), 26–47.

Meyer, A. R. and K. Sieber (1988). Towards fully abstract semantics for local variables: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 191–203. ACM Press.

Milner, R. (1978). A theory of type polymorphism in programming languages. *Journal of Computer and System Science 17*(3), 348–375.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. The MIT Press.

Mitchell, J. C. (1984). Coercion and type inference. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 175–185. ACM Press.

Mitchell, J. C. (1991). On the equivalence of data representations. In V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 305–330. Academic Press.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. MIT Press.

Mitchell, J. C. and E. Moggi (1991). Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic 51*(1–2), 99–124.

Mitchell, J. C. and G. D. Plotkin (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems 10*(3), 470–502.

Mitchell, J. C. and A. Scedrov (1993). Notes on sconing and relators. In E. Börger, G. Jäger, H. K. Büning, S. Martini, and M. M. Richter (Eds.), *Computer Science Logic '92, Selected Papers*, Volume 702 of *Lecture Notes in Computer Science*, pp. 352–378. Springer.

Mitchell, J. C. and R. Viswanathan (1996). Effective models of polymorphism, subtyping and recursion (extended abstract). In F. Meyer auf der Heide and B. Monien (Eds.), *23rd International Colloquium on Automata, Languages and Programming*, Volume 1099 of *Lecture Notes in Computer Science*, pp. 170–181. Springer.

Moggi, E. and A. Sabry (2004). An abstract monadic semantics for value recursion. *Theoretical Informatics and Applications 38*(4), 375–400.

Morrisett, G., A. Ahmed, and M. Fluet (2005). L3: A linear language with locations. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA '05)*, Volume 3461 of *Lecture Notes in Computer Science*. Springer.

O'Hearn, P. W. and D. J. Pym (1999). The logic of bunched implications. *Bulletin of Symbolic Logic 5*(2), 215–244.

O'Hearn, P. W. and J. C. Reynolds (2000). From algol to polymorphic linear lambda-calculus. *Journal of the ACM 47*(1), 167–223.

O'Hearn, P. W., J. C. Reynolds, and H. Yang (2001). Local reasoning about programs that alter data structures. In L. Fribourg (Ed.), *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, Volume 2142 of *Lecture Notes in Computer Science*, pp. 1–18. Springer.

O'Hearn, P. W. and R. D. Tennent (1995). Parametricity and local variables. *Journal of the ACM 42*(3), 658–709.

O'Hearn, P. W. and R. D. Tennent (Eds.) (1997). *Algol-Like Languages, Vols I and II*. Progress in Theoretical Computer Science. Birkhauser.

Oles, F. J. (1982). *A Category-theoretic approach to the semantics of programming languages*. Ph. D. thesis, Syracuse University.

Paulson, L. C. (1987). *Logic and Computation : Interactive proof with Cambridge LCF*, Volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Peyton Jones, S. (Ed.) (2003). *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press.

Phoa, W. (1992). An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, Department of Computer Science, University of Edinburgh.

Pierce, B. C. (1991). *Basic Category Theory for Computer Scientists*. MIT Press.

Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press.

Pierik, C. and F. S. de Boer (2005). A proof outline logic for object-oriented programming. *Theoretical Computer Science 343*(3), 413–442.

Pitts, A. M. (1996). Relational properties of domains. *Information and Computation 127*, 66–90.

Pitts, A. M. and I. D. B. Stark (1993). Observable properties of higher order functions that dynamically create local names, or: What's new? In A. M. Borzyszkowski and S. Sokolowski (Eds.), *Proceedings 18th International Symposium on Mathematical Foundations of Computer Science*, Volume 711 of *Lecture Notes in Computer Science*, pp. 122–141. Springer.

Pitts, A. M. and I. D. B. Stark (1998). Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts (Eds.), *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pp. 227–273. Cambridge University Press.

Plotkin, G. D. (1983). Domain theory. Pisa notes.

Poetzsch-Heffter, A. and P. Müller (1999). A programming logic for sequential Java. In S. D. Swierstra (Ed.), *European Symposium on Programming*, Volume 1576 of *Lecture Notes in Computer Science*, pp. 162–176. Springer.

Pym, D. J., P. W. O'Hearn, and H. Yang (2004). Possible worlds and resources: the semantics of BI. *Theoretical Computer Science 315*(1), 257–305.

Reddy, U. S. (1988). Objects as closures: Abstract semantics of object-oriented languages. In J. Chailloux (Ed.), *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 289–297. ACM Press.

Reddy, U. S. (1996). Global state considered unnecessary: An introduction to object-based semantics. *LISP and Symbolic Computation 9*(1), 7–76.

Reddy, U. S. (2002). Objects and classes in algol-like languages. *Information and Computation 172*(1), 63–97.

Reddy, U. S. and H. Yang (2004). Correctness of data representations involving heap data structures. *Science of Computer Programming 50*(1–3), 129–160.

Reus, B. (2002). Class-based versus object-based: A denotational comparison. In H. Kirchner and C. Ringeissen (Eds.), *Proceedings of 9th International Conference on Algebraic Methodology And Software Technology*, Volume 2422 of *Lecture Notes in Computer Science*, pp. 473–488. Springer.

Reus, B. (2003). Modular semantics and logics of classes. In M. Baatz and J. A. Makowsky (Eds.), *Computer Science Logic*, Volume 2803 of *Lecture Notes in Computer Science*, pp. 456–469. Springer.

Reus, B. and J. Schwinghammer (2004). Denotational semantics for Abadi and Leino's logic of objects. Technical Report 2004:03, Informatics, University of Sussex.

Reus, B. and J. Schwinghammer (2005). Denotational semantics for Abadi and Leino's logic of objects. In M. Sagiv (Ed.), *Proceedings of the European Symposium on Programming*, Volume 3444 of *Lecture Notes in Computer Science*, pp. 264–279. Springer.

Reus, B. and J. Schwinghammer (2006a). Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science 16*(2), 313–358.

Reus, B. and J. Schwinghammer (2006b). Separation logic for higher-order store. In *Proceedings Computer Science Logic (CSL'06)*, Lecture Notes in Computer Science. Springer. To appear.

Reus, B. and T. Streicher (2002). Semantics and logic of object calculi. In *Proceedings of 17th Annual IEEE Symposium Logic in Computer Science*, pp. 113–124. IEEE Computer Society Press.

Reus, B. and T. Streicher (2004). Semantics and logic of object calculi. *Theoretical Computer Science 316*, 191–213.

Reus, B., M. Wirsing, and R. Hennicker (2001). A Hoare-Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In H. Hussmann (Ed.), *FASE 2001*, Volume 2029 of *Lecture Notes in Computer Science*, pp. 300–317. Springer.

Reynolds, J. (1974). On the relation between direct and continuation semantics. In J. Loeckx (Ed.), *Automata, Languages and Programming*, Volume 14 of *Lecture Notes in Computer Science*, pp. 141–156. Springer.

Reynolds, J. C. (1978). Syntactic control of interference. In *Conference Record 5th ACM Symposium on Principles of Programming Languages*, pp. 39–46. ACM Press. Reprinted in (O'Hearn and Tennent 1997).

Reynolds, J. C. (1980). Using category theory to design implicit conversions and generic operators. In N. D. Jones (Ed.), *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, Number 94 in Lecture Notes in Computer Science, pp. 211–258. Springer.

Reynolds, J. C. (1981). The essence of Algol. In J. W. deBakker and J. C. van Vliet (Eds.), *Algorithmic Languages*, pp. 345–372. North-Holland.

Reynolds, J. C. (1982). Idealized Algol and its specification logic. In D. Néel (Ed.), *Tools and Notions for Program Construction*, pp. 121–161. Cambridge University Press.

Reynolds, J. C. (1983). Types, abstraction, and parametric polymorphism. In R. E. A. Mason (Ed.), *Information Processing 83*, pp. 513–523. Elsevier.

Reynolds, J. C. (2002a). Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pp. 55–74. IEEE Computer Society.

Reynolds, J. C. (2002b). What do types mean? — From intrinsic to extrinsic semantics. In A. McIver and C. Morgan (Eds.), *Essays on Programming Methodology*. Springer.

Schwinghammer, J. (2005a). A typed semantics for languages with higher-order store and subtyping. Technical Report 2005:05, Informatics, University of Sussex.

Schwinghammer, J. (2005b). A typed semantics of higher-order store and subtyping. In *Proceedings Ninth Italian Conference on Theoretical Computer Science*, Volume 3701 of *Lecture Notes in Computer Science*, pp. 390–404. Springer. To appear.

Scott, D. S. (1972). Continuous lattices. In F. W. Lawvere (Ed.), *Toposes, Algebraic Geometry and Logic*, Volume 274 of *Lecture Notes in Mathematics*, pp. 97–136. Springer.

Scott, D. S. (1993). A type-theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science 121*(1/2), 411–440. Reprint of a manuscript written in 1969.

Scott, D. S. and C. Strachey (1971). Toward a mathematical semantics for computer languages. In J. Fox (Ed.), *Proceedings Symposium Computers and Automata*. Polytechnic Inst. of Brooklyn Press. Also Technical Monograph PRG-6, Programming Research Group, Oxford University.

Shinwell, M. R. (2005). *The Fresh Approach: functional programming with names and binders*. Ph. D. thesis, University of Cambridge Computer Laboratory.

Shinwell, M. R. and A. M. Pitts (2005). On a monadic semantics for freshness. *Theoretical Computer Science 342*(1), 28–55.

Sieber, K. (1994). Full abstraction for the second order subset of an algol-like language. In I. Prívara, B. Rovan, and P. Ruzicka (Eds.), *Proceedings 19th International Symposium Mathematical Foundations of Computer Science*, Volume 841 of *Lecture Notes in Computer Science*, pp. 608–617. Springer.

Smyth, M. B. and G. D. Plotkin (1982). The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing 11*(4), 761–783.

Stark, I. (1996). Categorical models for local names. *LISP and Symbolic Computation 9*(1), 77–107.

Stark, I. (1998). Names, equations, relations: Practical ways to reason about *new*. *Fundamenta Informaticae 33*(4), 369–396.

Stevens, P. and R. Pooley (2000). *Using UML Software Engineering with Objects and Components* (updated ed.). Object Technology Series. Addison-Wesley.

Strachey, C. (1966). Towards a formal semantics. In T. B. Steel (Ed.), *Formal Language Description Languages for Computer Programming*, pp. 198–220. North-Holland.

Stroustrup, B. (2000). *The C++ Programming Language* (Third ed.). Addison-Wesley.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys 28*(3), 438–479.

Talcott, C. L. (1998). Reasoning about functions with effects. In A. D. Gordon and A. M. Pitts (Eds.), *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pp. 347–390. Cambridge University Press.

Tang, F. and M. Hofmann (2002). Generation of verification conditions for Abadi and Leino's logic of objects. Presented at 9th International Workshop on Foundations of Object-Oriented Languages.

Tennent, R. D. (1985). Functor-category semantics of programming languages and logics. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard (Eds.), *Category Theory and Computer Science*, Volume 240 of *Lecture Notes in Computer Science*, pp. 206–224. Springer.

Tennent, R. D. and D. R. Ghica (2000). Abstract models of storage. *Higher-Order and Symbolic Computation 13*(1–2), 119–129.

Thorup, L. and M. Tofte (1994). Object-oriented programming and standard ML. In *Record of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*.

Viswanathan, R. (1998). Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press.

von Oheimb, D. (2001). Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience 13*(13), 1173–1214.

Winskel, G. (1993). *The Formal Semantics of Programming Languages*. MIT Press.

Wright, A. K. (1995). Simple imperative polymorphism. *LISP and Symbolic Computation 8*(4), 343–355.

Yang, H. (2001). An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*.

Zhang and Nowak (2003). Logical relations for dynamic name creation. In *17th Workshop on Computer Science Logic (CSL'03)*, Volume 2803 of *Lecture Notes in Computer Science*, pp. 575–588. Springer.