Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

Step-indexed Semantic Model of Types for the Functional Object Calculus

submitted by

Cătălin Hrițcu

on May 16, 2007

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dr. Jan Schwinghammer

Reviewers

Prof. Dr. Gert Smolka

Prof. Dr. Holger Hermanns

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Cătălin Hriţcu
Saarbrücken, May 16, 2007

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Cătălin Hriţcu
Saarbrücken, May 16, 2007

Abstract

Step-indexed semantic models of types were proposed as an alternative to the purely syntactic proofs of type safety using subject-reduction. This thesis introduces a step-indexed model for the functional object calculus, and uses it to prove the soundness of an expressive type system with object types, subtyping, recursive and bounded quantified types.

Acknowledgements

I am profoundly indebted to my advisor, Jan Schwinghammer, for his continuous support during the last year. I thank Jan for his unlimited patience when explaining the basic concepts I needed to know before I could get started with this thesis. Not only did he share with me some of his knowledge and ideas, but also some of his enthusiasm. Our discussions provided much motivation, and radically changed the way I perceive theoretical computer science.

I am grateful to Gert Smolka for his valuable advice given on many occasions, for his inspiring introductions to logic and semantics, and for offering me the opportunity to work with Jan. I also thank Gert Smolka and Holger Hermanns for reviewing this thesis.

Finally, I would like to thank the members of the Programming Systems Lab and the International Max Planck Research School for Computer Science for their friendship and support.

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove." – Antoine de Saint-Exupery

Contents

Chapter 1

Introduction

In this thesis we prove the soundness of an expressive type system, i.e. the fact that well-typed terms do not produce type errors when evaluated.

The most common way to prove a type system sound is by a purely syntactic proof technique called subject-reduction, which was adapted from combinatory logic by Wright and Felleisen [WF94]. In this syntactic setting the typing judgement is a relation that is recursively defined by the typing rules. One shows that reduction preserves the typability of terms (preservation) and that well-typed terms that are not yet values can always be reduced (progress).

This is not the only way though. The soundness of a type system can also be proved with respect to a semantic model. One defines a meaning function relating the syntactic typing judgement to a semantic typing judgement. One proves that whenever the syntactic judgement holds the corresponding semantic judgement is also valid (semantic soundness). Then one shows in the model that the semantic judgement enforces type soundness. Usually the model is denotational using for instance ordered sets. Constructing such a model and proving the desired properties tends to be involved.

This thesis shows that it is possible to use much simpler semantic models constructed from the operational semantics by a technique called *step-indexing*. Our long-term goal is to use this technique in proving the soundness of more powerful deduction systems such as program logics, for which purely syntactic arguments are not always applicable.

1.1 Step-indexed Semantic Models

Step-indexed semantic models were introduced by Appel and his collaborators in the context of foundational proof-carrying code as an alternative to subject-reduction. The goal was to construct more elementary and more modular type soundness proofs which can be easier checked automatically [AF00, AM01].

Step-indexed models are based only on the small-step operational semantics of an untyped calculus. Semantic types are defined just as sets of values indexed by natural numbers. Intuitively, a term has a certain semantic type if it behaves like an element of that type for any number of computation steps.

Syntax

$$\begin{aligned} a, b &::= x \mid \lambda x.\, b \mid a\ b \mid \mu(x)\,\{f_d{=}b_d\}_{d\in D} \mid a.f \mid \ldots && \textit{(Terms)} \\ v &::= \lambda x.\, b \mid \mu(x)\,\{f_d{=}b_d\}_{d\in D} \mid \text{true} \mid \text{false} \mid \underline{0} \mid \underline{1} \mid \underline{2} \mid \ldots && \textit{(Values)} \end{aligned}$$

Operational Semantics (Small-step)

$$\begin{aligned} (\lambda x.\, b)\ v &\to [x \mapsto v]\,(b) && \textit{(}\beta\textit{)} \\ \mu(x)\,\{f_d{=}b_d\}_{d\in D}\,.f_e &\to \left[x \mapsto \mu(x)\,\{f_d{=}b_d\}_{d\in D}\right](b_e), \text{ if } e \in D && \textit{(}\mu\textit{)} \end{aligned}$$

Figure 1.1: λ-calculus with booleans, numbers and recursive records

1.1.1 An Illustrative Example

In order to give more intuition about step-indexed semantic models we sketch such a model for the call-by-value λ-calculus extended with booleans, natural numbers and recursive records[1]. This model is a simplification of the one for general recursive types by Appel and McAllester [AM01]. It is also the base of our step-indexed model for the functional object calculus, which we present more formally in Chapter 3.

The syntax and small-step operational semantics of the untyped lambda calculus we consider in this example form the base of the step-indexed model. The parts which are interesting here are given in Figure 1.1.

Semantic types are sets of indexed values. A pair $\langle k, v\rangle$ is in some type τ, if one cannot distinguish v from a "real" value of type τ in less than k computation steps. For example $\langle 1, \lambda x.\,\text{true}\rangle$ is in the type $\mathit{Bool} \to \mathit{Nat}$ since in one step we can only apply the function, but then there are no steps left to observe that true is not an integer. On the other hand $\langle 2, \lambda x.\,\text{true}\rangle$ is not in $\mathit{Bool} \to \mathit{Nat}$ because two steps are already enough to distinguish $\lambda x.\,\text{true}$ from a real $\mathit{Bool} \to \mathit{Nat}$ function: one step for applying the function to some argument, and the other step for attempting to perform some integer operation on the boolean "true" and failing. So we say that $\langle k, v\rangle \in \tau$, if every context of type τ (e.g. every function of type $\tau \to \alpha$) safely executes for at least k steps when applied to v.

A semantic type is therefore built as a sequence of increasingly accurate approximations. In the end we are only interested in the limit of such a sequence since it contains the real values of the type, while each of the approximations also contains "false positives". However, the usefulness of building types based on such approximations becomes evident once any form of recursion is considered.

Based on the approximate types of values we can also express that a closed term behaves like an element of a type for a number of computation steps. A closed term a has a type τ for k steps (which we denote as $a :_k \tau$), if whenever a reduces to an irreducible term b in $j < k$ steps, then b is a value of type τ for the remaining $k - j$ computation steps. For example $\lambda x.\,\text{true} :_1 \mathit{Bool} \to \mathit{Nat}$ since functions are values thus irreducible, and we have already seen that $\langle 1, \lambda x.\,\text{true}\rangle \in \mathit{Bool} \to \mathit{Nat}$. Similarly, $(\lambda x.\, x)\ (\lambda x.\,\text{true}) :_2 \mathit{Bool} \to \mathit{Nat}$ since $(\lambda x.\, x)\ (\lambda x.\,\text{true})$ reduces in one β step to $\lambda x.\,\text{true}$. Also any closed term that safely reduces for at least two steps has any type for two steps, for example

[1]Recursive records are closely related to object calculi without method updates [KR94, BCP99].

$(\lambda x.\, x)\ ((\lambda x.\, x)\ \text{true}) :_2 \mathit{Bool} \to \mathit{Nat}$. On the other hand, none of these statements hold if we increase the approximation index by one, so these terms are “false positives”.

With this setup in place we can define each of our semantic types. The base types *Bool* and *Nat* are completely described by the set of their values. A value is in a base type either for every index or not at all.

$$\begin{aligned}\text{Bool} &\triangleq \{\langle k, v\rangle \mid k \in \mathbb{N}, v \in \{\text{true}, \text{false}\}\}\\ \text{Nat} &\triangleq \{\langle k, \underline{n}\rangle \mid k, n \in \mathbb{N}\}\end{aligned}$$

In general, a function has type $\alpha \to \beta$ if when invoked with an argument of type α produces a result of type β. In the step-indexed model, a function has type $\alpha \to \beta$ for k computation steps if when applied to any well-typed argument of type α it produces a result that has type β for another $k - 1$ steps. This because the application itself takes one computation step, and the only way to use a function is by applying it to some argument. We also have to take into account that the function can be applied after some computation steps, so for every $j < k$ when applying the function to a value in type α for j steps, the result must have type β for j steps.

$$\alpha \to \beta \triangleq \{\langle k, \lambda x.\, b\rangle \mid \forall j{<}k.\ \forall v.\ \langle j, v\rangle \in \alpha \Rightarrow [x \mapsto v]\,(b) :_j \beta\}$$

Recursive records are collections of fields that can recursively reference the record containing them. Selecting a field will substitute all occurrences of the “self-reference” variable by the target record. Once a record is created its fields can no longer be updated, since updates do not interact well with recursion.

The recursive record type $\{f_d : F_d\}_{d\in D}$ associates to each field f_d a function F_d taking the type of the whole record as argument, and returning the type of the field. This formulation can lead to direct circularity, since a recursive record can directly contain itself and its type needs to reflect this. For example we expect a record with a single field referencing the record containing it: $\mu(x)\,\{f{=}x\}$, to have type $\{f : (\lambda\alpha{\in}\mathsf{Type}.\, \alpha)\}$, that is the type of records with a single field f that has the same type as the record containing it. This can only happen if the type satisfies the following recursive equation:

$$\{f : (\lambda\alpha{\in}\mathsf{Type}.\, \alpha)\} = \{f : \{f : (\lambda\alpha{\in}\mathsf{Type}.\, \alpha)\}\}$$

Step-indexing is extremely useful here, since it can make the recursive definition of record types well-founded. A recursive record $\mu(x)\,\{f_d{=}b_d\}_{d\in D}$ has type $\{f_d : F_d\}_{d\in D}$ for k steps if the term obtained by selecting one of its fields f_d has type $F_d(\{f_d : F_d\}_{d\in D})$ for any number of steps strictly smaller than k.

$$\begin{aligned}\{f_d : F_d\}_{d\in D} \triangleq \{\langle k, \mu(x)\,\{f_d{=}b_d\}_{d\in D}\rangle \mid{}& \forall d{\in}D.\ \forall j{<}k.\\ & \left[x \mapsto \mu(x)\,\{f_d{=}b_d\}_{d\in D}\right](b_d) :_j F_d(\{f_d : F_d\}_{d\in D})\}\end{aligned}$$

Independent of the particular semantic types we define the semantic type judgement: $\Sigma \models a : \tau$ holds for a possibly open term a, if after substituting any well-typed values for the free variables of a we obtain a closed term that has type τ for any number of computation steps. This definition directly enforces that all terms that are typable using the semantic typing judgement do not produce type errors when evaluated.

$$(\textsc{Var})\ \Sigma \models x : \Sigma(x) \qquad (\textsc{Add})\ \frac{\Sigma \models a : \mathit{Nat} \quad \Sigma \models b : \mathit{Nat}}{\Sigma \models a + b : \mathit{Nat}}$$

$$(\textsc{Lam})\ \frac{\Sigma[x \mapsto \alpha] \models b : \beta}{\Sigma \models \lambda x.\, b : \alpha \to \beta} \qquad (\textsc{App})\ \frac{\Sigma \models a : \beta \to \alpha \quad \Sigma \models b : \beta}{\Sigma \models a\, b : \alpha}$$

$$(\textsc{Rec})\ \frac{\forall d{\in}D.\ \Sigma\big[x \mapsto \mu(x)\,\{f_d{=}b_d\}_{d\in D}\big] \models b_d : F_d(\{f_d : F_d\}_{d\in D})}{\Sigma \models \mu(x)\,\{f_d{=}b_d\}_{d\in D} : \{f_d : F_d\}_{d\in D}}$$

$$(\textsc{Sel})\ \frac{\Sigma \models a : \{f_d : F_d\}_{d\in D}}{\Sigma \models a.f_e : F_d(\{f_d : F_d\}_{d\in D})}$$

Figure 1.2: Some of the semantic typing rules

Since the semantic judgement is defined independently of any typing rules, we would have to prove individual typing judgements directly from the definitions of their corresponding types. For example we could prove that $\emptyset \models \lambda x.\, \lambda y.\, x + y : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$ or that $[x \mapsto \mathit{Nat}]\,[y \mapsto \mathit{Nat}] \models x + y : \mathit{Nat}$. However, this would be tedious and a lot of work would be duplicated between similar derivations. In order to avoid this duplication, we can prove general lemmas which we call semantic typing rules only once and then use them to build type derivations in the usual way.

$$(\textsc{Lam})\ \dfrac{(\textsc{Lam})\ \dfrac{(\textsc{Add})\ \dfrac{(\textsc{Var})\ \dfrac{}{[x \mapsto \mathit{Nat}]\,[y \mapsto \mathit{Nat}] \models x : \mathit{Nat}} \qquad \dfrac{}{[x \mapsto \mathit{Nat}]\,[y \mapsto \mathit{Nat}] \models y : \mathit{Nat}}\ (\textsc{Var})}{[x \mapsto \mathit{Nat}]\,[y \mapsto \mathit{Nat}] \models x + y : \mathit{Nat}}}{[x \mapsto \mathit{Nat}] \models \lambda y.\, x + y : \mathit{Nat} \to \mathit{Nat}}}{\emptyset \models \lambda x.\, \lambda y.\, x + y : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}}$$

Figure 1.2 shows some of the semantic rules we can prove for our calculus.

The semantic typing judgements have a semantic truth value associated with them, and the semantic typing rules allow us to derive true judgements using other true judgements as premises. One proves that the semantic typing rules are sound, that is if the premises of a rule are true then the conclusion is also true. This is done either directly from the definitions or by induction on the index (e.g. Rec).

Appel et. al. use the semantic typing rules directly for type-checking programs [AF00, AM01, ARS02, AAV03], as in the simple example above. However, when one considers more complex type systems with subtyping, polymorphism or recursive types the semantic typing rules no longer directly correspond to their syntactic counterparts. This leads to more complex models (like the one developed by Swadi to track type variables [Swa03]) and to undecidable type-checking, which cannot be avoided in their particular setting.

In this thesis we take a different approach, and use the semantic typing rules only to prove the semantic soundness of a syntactic type system, which can be made decidable and is thus more suitable for type checking programs.

1.2 The Functional Object Calculus

The language considered in this thesis is the functional object calculus, a very expressive, yet extremely simple object-oriented programming language. It was introduced by Abadi and Cardelli for investigating the theoretical properties of objects, and in particular for studying their type systems [AC96a].

The functional object calculus models all the important aspects of widely-used object-oriented programming languages, as well as the the often subtle interactions between them. It can express objects, subtyping, classes, inheritance, parametric polymorphism and other features of programming languages such as Java and C#.

At the same time the functional object calculus is amazingly simple, and captures the essence of object-oriented programming languages in pretty much the same way the lambda calculus captures the essence of the functional ones. This allows one to focus on relevant object-oriented concepts in their most general form, by abstracting away from the design decisions, implementation details, peculiarities and mistakes present in real object-oriented languages.

1.3 Outline

In **Chapter 2** we present the untyped functional object calculus introduced by Abadi and Cardelli under the name of ς-calculus [AC96b, AC96a]. This very simple calculus has only objects, so we also present encodings for procedures, booleans, natural numbers, a fixpoint operator, classes and inheritance.

In **Chapter 3** we construct purely semantic types for the ς-calculus. The safety of well-typed terms is immediate, and each of the typing rules is proved sound with respect to the model. The step-indexed semantic model we present extends the one for recursive types of Appel and McAllester [AM01] with object types, subtyping and bounded quantified types.

We start with a simple definition of object types that we then extend to accommodate subtyping. For a stronger notion of subtyping we restrict certain method invocations and updates which leads to object types with variance annotations. We also study the interaction between subtyping and recursive and quantified types in a step-indexed model.

The semantic type system from Chapter 3 is sound but undecidable. In **Chapter 4** we define the usual syntactic type expressions which also contain type variables, we extend the ς-calculus with type annotations, we give a syntactic type system for the modified calculus for which type checking can be made decidable. We then prove the semantic soundness of the syntactic type system with respect to the semantic model from Chapter 3, which immediately implies that well-typed annotated terms are safe to evaluate after type erasure.

Finally, **Chapter 5** draws conclusions, presents related work and and introduces several interesting directions for future work.

Chapter 2

The Untyped ς-calculus

The functional object calculus (ς-calculus) is a very simple object-oriented calculus introduced by Abadi and Cardelli in [AC96b], and then presented in their book entitled *A Theory of Objects* [AC96a, Chapter 6]. It only has one primitive: objects. Still, many other constructs can be conveniently encoded, including procedures and classes.

The calculus presented in this chapter is *untyped*. However, in Chapter 3 we introduce purely semantic types for it, and in Chapter 4 we extend the ς-calculus with type annotations and present a syntactic type system for it.

2.1 Syntax

We start with the simplest possible calculus and pack up more features as we go, mostly as encodings. The ς-calculus has only four syntactic forms: variables, objects, method invocations and method updates. The abstract syntax of ς-terms is formalized by the following grammar.

$$a, b \in \mathsf{Ter} ::= x \mid [m_d{=}\varsigma(x_d)b_d]_{d\in D} \mid a.m \mid a.m := \varsigma(x)b$$

The meta-variables a and b range over ς-terms, and x ranges over a countably infinite set of variables.

Objects are collections of named methods. Each method $\varsigma(x)b$ has a *self argument* x, a bound variable that represents the object containing the method, and a method body b, an arbitrary term that will evaluate to the result of the method. The self argument can be used inside a method body for invoking the methods of the containing object[1], including the calling method itself which leads to direct recursion. We consider two terms to be syntactically equivalent if they are identical up to the consistent renaming of bound variables.

An object is created by providing all methods it will contain. The order in which these methods are considered does not matter. In the ς-calculus after an object is created, its methods can be invoked or updated, but no new methods can be added, and the existing methods cannot be deleted.

When the names and bodies of the methods are of no importance to our presentation, we will iterate over some finite set (for example D in $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$)

[1]The self argument has a similar role to the keyword "this" in Java and C++.

Let $v \equiv [m_d{=}\varsigma(x_d)b_d]_{d\in D}$

(Red-Inv) $v.m_e \rightarrow [x_e \mapsto v]\,(b_e)$, where $e \in D$

(Red-Upd) $v.m_e \coloneqq \varsigma(x)b \rightarrow [m_e{=}\varsigma(x)b, m_d{=}\varsigma(x_d)b_d]_{d\in D\setminus\{e\}}$, where $e \in D$

(Red-Ctx) $\dfrac{a \rightarrow b}{C[a] \rightarrow C[b]}$, where $C[\bullet] ::= \bullet \mid C.m \mid C.m \coloneqq \varsigma(x)b$

Figure 2.1: Small-step operational semantics of the ς-calculus

to generate names for the methods (m_d), the self variables (x_d) and the method bodies (b_d).

The syntax of the ς-calculus does not provide fields directly, but they can be simulated by constant methods that do not use the self argument. Moreover, we omit unused ς binders: whenever x does not appear in b we abbreviate $\varsigma(x)b$ as just b. This basically allows fields as syntactic sugar, and since methods are not terms and can only appear inside an object creation or a method update this notation does not lead to ambiguities.

Also please note that in the syntax of terms, methods do not have arguments other than self. However, there are straightforward ways to emulate methods with arguments. The caller of a method can store the arguments in some fields of the invoked object prior to the method call, and the invoked method can access these arguments using self. As explained later, a more systematic way to accommodate methods with arguments is to use procedures as method bodies.

The only *values* in the ς-calculus are the objects.

$$v \in \mathsf{Val} ::= [m_d{=}\varsigma(x_d)b_d]_{d\in D}$$

In this thesis we often consider only values without free variables, which we call *closed values*.

$$\mathsf{CVal} \triangleq \{v \in \mathsf{Val} \mid FV(v) = \emptyset\}$$

2.2 Operational Semantics

The small-step operational semantics of the ς-calculus is defined by the rules in Figure 2.1.

Method invocation is defined as self substitution. If v is an object value $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$, then a method invocation $v.m_e$ reduces to the body of the method b_e, where all references to the self argument x_e are substituted with the host object v. By $[x \mapsto a]\,(b)$ we denote the result of the capture-avoiding substitution of x with a in b. Capture-avoiding substitution is formally defined in Appendix A.2.

Method updates have a functional semantics: a method update reduces to a copy of the target object where the updated method is replaced by a new one.

The evaluation strategy is defined using a context rule and reduction contexts. These enforce an evaluation strategy, which corresponds to call-by-name evaluation in the λ-calculus [Plo75]. No reduction can occur below a ς binder, so that the body of a method is evaluated every time the method is invoked. But since we encode fields as constant methods, fields are also evaluated every time they are selected. While this is not very common, it does not cause significant problems in this functional setting where the only side-effect is non-termination.

Example 2.2.1 (Non-termination)**.** As previously mentioned, a method can invoke itself recursively, and this can lead to non-termination. The following term directly reduces to itself:

$$[m{=}\varsigma(x)x.m].m \to [x \mapsto [m{=}\varsigma(x)x.m]]\,(x.m) \equiv [m{=}\varsigma(x)x.m].m \to \dots$$

The reduction relation in Figure 2.1 is deterministic: for each term there is at most one other term to which it can reduce. We call a term *irreducible* (denoted by $a \not\to$), if none of the reduction rules can be applied to it. Values are irreducible, but not all irreducible terms are values. We call a term that is irreducible yet not a value *stuck*. For example $[\,].m$ is a stuck term since the invoked method m does not exist in the empty object. We also define multi-step reduction and evaluation in the usual way [BN98]: $\to^0 \triangleq \{(a,a) \mid a \in \mathsf{Ter}\}$, $\to^{n+1} \triangleq \to^n \circ \to$ and $\to^* \triangleq \bigcup_{n\geq 0} \to^n$

2.3 Encodings

2.3.1 Call-by-name λ-calculus

The untyped call-by-name λ-calculus can be encoded into the ς-calculus by a simple transformation [AC96a, Section 6.3.2]. Procedures (also called lambda abstractions or functions) are transformed into objects having a field for the argument (*arg*), and a method for the body of the procedure (*eval*). In the body of the procedure all references to the argument are replaced by selections of the field through the self object. The application of such an encoded procedure is done by first updating the field to the actual argument, and then invoking the method that accesses it through self.

$$\langle\!\langle x\rangle\!\rangle \triangleq x$$

$$\langle\!\langle \lambda x.\, b\rangle\!\rangle \triangleq [\mathit{arg}{=}\varsigma(x)x.\mathit{arg},\, \mathit{eval}{=}\varsigma(x)\,[x \mapsto x.\mathit{arg}]\,(\langle\!\langle b\rangle\!\rangle)]$$

$$\langle\!\langle a\ b\rangle\!\rangle \triangleq (\langle\!\langle a\rangle\!\rangle\,.\mathit{arg} \coloneqq \langle\!\langle b\rangle\!\rangle).\mathit{eval}$$

Because of our lazy semantics of fields, the argument of an encoded procedure is evaluated every time it is used, as in call-by-name. In this simple setting without any strict constructs and without continuations [Plo75], the call-by-value λ-calculus cannot be encoded. Furthermore, the encoding validates the β but not the η rule in the λ-calculus [GR96], and accommodating the usual typing rules for encoded procedures is not so easy in the presence of subtyping. This is the price we pay for keeping the ς-calculus as simple as possible, thus without primitive procedures or eager fields.

Even without having them as primitive, we will use procedures frequently as syntactic sugar in the remainder of this chapter, for example for modelling methods with arguments, as in $[m{=}\varsigma(x)\lambda y.\,\lambda z.\,b]$ where m is a method that has two arguments (y and z) other than self (x).

2.3.2 Booleans

Having encoded procedures also allows us to use the Church encoding for booleans.

$$\text{true} \triangleq \lambda x.\, \lambda y.\, x \qquad \text{false} \triangleq \lambda x.\, \lambda y.\, y$$

For readability we also define an if-then-else construct.

$$\text{if } b \text{ then } a_1 \text{ else } a_2 \triangleq b\ a_1\ a_2$$

2.3.3 Natural Numbers

While we could also define natural numbers using their Church encoding, the direct encoding into objects is more intuitive [AC96a, Section 6.5.3]. Numbers are objects with an *iszero* boolean field and two methods yielding the preceding (*pred*) and the succeeding number (*succ*). We only need to construct a term for zero, as all the other numbers are its successors. Zero is defined to have itself as the predecessor, and its *iszero* field is set to true. When the *succ* method is invoked for zero a new object corresponding to the number one is created. For this object the *iszero* field is set to false, *pred* returns the outer self argument representing zero in this case, while the *succ* method is copied verbatim from the predecessor.

$$\underline{0} \triangleq [\mathit{iszero}{=}\text{true}, \mathit{pred}{=}\varsigma(x)x, \mathit{succ}{=}\varsigma(x)[\mathit{iszero}{=}\text{false}, \mathit{pred}{=}x, \mathit{succ}{=}x.\mathit{succ}]]$$

$$\underline{n} \triangleq \underline{0}\underbrace{.\mathit{succ}.\ \ldots\ .\mathit{succ}}_{n \text{ times}}$$

Having this as a start we can define recursive procedures for sum, difference and product, which can then be used for defining other arithmetical operations. Below we give the implementation of factorial and greatest common divisor.

Example 2.3.1 (Factorial)**.**

$$[\mathit{fac} = \varsigma(y)\lambda n.\, \text{if } n.\mathit{iszero} \text{ then } \underline{1} \text{ else } n \times (y.\mathit{fac}\ (n.\mathit{pred}))].\mathit{fac}$$

Example 2.3.2 (Euclid GCD)**.**

$$\begin{aligned}[\mathit{gcd} = \varsigma(y)\lambda x.\, \lambda z.\, &\text{if } x{<}z \text{ then } y.\mathit{gcd}\ x\ (z{-}x)\\ &\quad\text{else if } z{<}x \text{ then } y.\mathit{gcd}\ (x{-}z)\ z \text{ else } x].\mathit{gcd}\end{aligned}$$

2.3.4 Fixpoint Operator

The previous two examples used exactly the same pattern to obtain a recursive procedure by direct recursion through self. We can capture this pattern into a general fixpoint operator for procedures [FHM94]:

$$\mathit{fix} \triangleq \lambda f.\, [\mathit{rec}{=}\varsigma(x)f\ (x.\mathit{rec})].\mathit{rec}$$

Using this fixpoint operator we can give a slightly simpler recursive definition of the factorial, that matches exactly the one from the λ-calculus:

$$\mathit{fac} = \mathit{fix}\ \lambda f.\, \lambda n.\, \text{if } n.\mathit{iszero} \text{ then } \underline{1} \text{ else } n \times (f\ (n.\mathit{pred}))]$$

2.3.5 Classes and Inheritance

Procedures are also used for representing *pre-methods*: procedures that become methods once they are embedded into objects. For us classes and traits are collections of mutually dependent pre-methods, while inheritance means pre-method reuse [AC96a, Section 6.6].

Traits are collections of pre-methods that are usually not self-contained, so it might take more than one trait to construct useful objects[2]. Traits are objects themselves where each pre-method is stored in a field, for example[3]: $[m_d{=}\lambda x_d.\, b_d]_{d\in D}$.

The union of several disjoint traits is simply the trait containing the pre-methods in all of them. For example given two disjoint sets D and E, and the traits $[m_d{=}\lambda x_d.\, b_d]_{d\in D}$ and $[m_e{=}\lambda x_e.\, b_e]_{e\in E}$, we can construct the union trait $[m_{d'}{=}\lambda x_{d'}.\, b_{d'}]_{d'\in D\cup E}$.

A trait is complete when all its pre-methods reference only pre-methods existing in that trait. From a complete trait we can directly construct objects. For example from $[m_d{=}\lambda x_d.\, b_d]_{d\in D}$ we can construct $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$.

A *class* is just a complete trait together with a *new* method. The *new* method constructs objects corresponding to the class by applying each pre-method to the self of the constructed object. For example $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$ could have been generated by the *new* method of the following class:

$$[new{=}\varsigma(y)\; [m_d{=}\varsigma(x_d)(y.m_d)\; x_d]_{d\in D}\;, m_d{=}\lambda x_d.\, b_d]_{d\in D}$$

As mentioned earlier, in this setting *inheritance* means reusing pre-methods from existing traits and classes. For example if $t \equiv [m_d{=}\lambda x_d.\, b_d]_{d\in D}$, then the trait $[m_d{=}t.m_d, m_e{=}\lambda x_e.\, b_e]_{d\in D, e\in E}$ inherits all methods in t, and adds the new methods denoted by m_e. In a similar way if $D = D' \uplus D''$ then the class

$$[new{=}\ldots, m_{d'}{=}t.m_{d'}, m_{d''}{=}\lambda x_{d''}.\, b_{d''}, m_e{=}\lambda x_e.\, b_e]_{d'\in D', d''\in D'', e\in E}$$

inherits from t the methods $m_{d'}$, overrides the methods $m_{d''}$, and adds the new methods m_e.

Multiple inheritance is possible by simultaneously inheriting pre-methods from more traits or classes. Please note that partially inheriting a trait or a class without overriding all the pre-methods that are not inherited is in general unsafe because of the possible dependencies between the pre-methods in the same trait or class.

Example 2.3.3 (Shapes)**.** Below we define a class *square* that inherits the field *pos* from the trait *shape*, adds a new field *side* and overrides the *area* method.

$$shape = [pos{=}[x{=}\underline{0}, y{=}\underline{0}], area{=}\underline{0}]$$

$$\begin{aligned} square = [&new{=}\varsigma(z)[pos{=}z.pos, side{=}z.side, area{=}\varsigma(x)(z.area)\; x],\\ &pos{=}shape.pos, side{=}\underline{0}, area{=}\lambda x.\, (x.side \times x.side)] \end{aligned}$$

Using the *square* class is straightforward, for example

$$(square.new.side \coloneqq \underline{10}).area$$

produces $\underline{100}$ as a result.

[2]Incomplete traits are similar to abstract classes in Java and C++.

[3]Please note that the binders in the trait are λ-binders, while the ς-binders have been elided according to our notation of fields.

While this encoding of classes, traits and inheritance is rather raw, many of its details can be hidden behind a more traditional syntax [AC96a, Chapter 12].

Chapter 3

Step-indexed Semantic Model of Types

3.1 The Semantic Model

Step-indexed semantic models were first developed by Appel and his collaborators in the context of foundational proof-carrying code [AF00]. Their goal was to construct proofs of type soundness for low-level languages, that are easier to machine-check than the widely-used subject-reduction proofs [WF94]. Their new technique proved to be general enough to be successfully applied to a pure λ-calculus with recursive types [AM01], and was then extended to general references, impredicative polymorphism [AAV03, Ahm04] and substructural state [AFM05]. Here we extend the step-indexed model introduced by Appel and McAllester [AM01] with object types, subtyping and bounded quantified types.

Step-indexed semantic models are based only on the small-step operational semantics of an untyped calculus. They tend to be much simpler than the corresponding denotational models based on complete partial orders, and the proofs are usually just by direct induction on the index.

Our semantic types are sets of closed values indexed by a number of remaining "safe" computation steps. Intuitively, a closed value v is in type α with approximation k (which we denote as $\langle k, v\rangle \in \alpha$), if one cannot distinguish v from a real value of type τ in less than k computation steps. As we will see, this indexing makes the recursive definitions of types well-founded.

Additionally we require that types are *closed under descending index*, corresponding to the intuition that if a value cannot be distinguished from a real value of a type in a certain number of steps, then it cannot be distinguished in any smaller number of steps.

Definition 3.1.1 (Semantic Types)**.** Types are sets of pairs formed of an index and a closed value, with the requirement that these sets are closed under descending index.

$$\mathsf{Type} = \{\tau \subseteq \mathbb{N} \times \mathsf{CVal} \mid \forall k{\geq}0.\ \forall j{\leq}k.\ \forall v{\in}\mathsf{CVal}.\ \langle k, v\rangle \in \tau \Rightarrow \langle j, v\rangle \in \tau\}$$

We will use α, β and τ to denote sets of indexed values, and if that is the case, we explicitly require them to be types. We call $\lfloor\tau\rfloor_k$ the k-th approximation of the set τ and define it as the subset containing all elements of τ that have an index that is strictly less than k.

Definition 3.1.2 (Semantic Approximation)**.**

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

Definition 3.1.3 (Top and Bottom)**.** The empty set ($\bot$) and the set of all value-index pairs ($\top$) trivially satisfy the closure under descending index condition, so they are types.

$$\begin{aligned}\bot &\triangleq \emptyset\\ \top &\triangleq \{\langle k, v\rangle \mid k \in \mathbb{N}, v \in \mathsf{CVal}\}\end{aligned}$$

We do not have other base types in the ς-calculus (they would be trivial to represent, see introduction).

Typing a closed value corresponds to testing whether it belongs to a type for all approximation indices. However, we want to type not only closed values, but also closed terms. A closed term a has a type τ with approximation k (which we denote as $a :_k \tau$), if a behaves like an element of τ for k computation steps. More precisely, if a closed term a satisfies $a :_k \tau$ and it reduces to an irreducible term b in $j < k$ steps, then it should be the case that b is a value in τ for the remaining $k - j$ computation steps.

Definition 3.1.4 (ClosedTerm:$_k$Type)**.** If a such that $FV(a) = \emptyset$, then

$$a :_k \tau \;:\Leftrightarrow\; \forall j{<}k.\ (a \rightarrow^j b \;\wedge\; b \not\rightarrow) \Rightarrow \langle k{-}j, b\rangle \in \tau$$

We state two simple properties that are often used in the proofs. First, if a closed term has a type with approximation k then it also has this type with any approximation smaller than k.

Property 3.1.5. *If $a :_k \tau$ and $j \leq k$ then $a :_j \tau$.*

Proof. Immediate from Def. 3.1.1 and 3.1.4. □

Second, a closed value v has a type τ with approximation $k > 0$, if and only if the pair $\langle k, v\rangle$ belongs to the set τ.

Property 3.1.6. $\forall k > 0.\ \forall v \in \mathit{CVal}.\ v :_k \tau \Leftrightarrow \langle k, v\rangle \in \tau$

Proof. Immediate from Def. 3.1.4 since values are irreducible. □

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

Definition 3.1.7 (ValueEnv:$_k$TypeEnv)**.** A *semantic type environment* is a finite map from variables to types. A *value environment* is a finite map from variables to closed values. We say that a value environment σ agrees with a semantic type environment Σ with approximation k (which we denote as $\sigma :_k \Sigma$) if for all variables x in the domain of Σ we have that $\sigma(x) :_k \Sigma(x)$.

$$\sigma :_k \Sigma \;:\Leftrightarrow\; \forall x \in \mathit{Dom}(\Sigma).\ \sigma(x) :_k \Sigma(x)$$

The semantic typing judgement is then defined by quantifying over all approximation indices and all approximately well-typed value environments.

Definition 3.1.8 (Semantic Typing Judgement)**.** A term a has type τ with respect to a semantic type environment Σ, if all free variables in a are mapped by Σ and for all approximation indices k, and all value environments σ that agree with Σ with approximation k, we have that the closed term obtained by applying the substitution σ to a has type τ with approximation k.

$$\Sigma \models a : \tau \;:\Leftrightarrow\; \mathit{FV}(a) \subseteq \mathit{Dom}(\Sigma) \;\wedge\; \forall k{\geq}0.\ \forall \sigma :_k \Sigma.\ \sigma(a) :_k \tau$$

The semantic typing judgements have a meaning, a semantic truth value associated with them, and the semantic typing rules we will introduce and prove will allow us to derive true judgements using other true judgements as premises. But before we introduce any rules, we show that all terms that are typable using the semantic typing judgement are also safe to execute. Together with the soundness of all the rules this will ensure the type soundness of our semantic type system.

Definition 3.1.9 (Safe for k Steps)**.** We call a term *safe for k steps*, if it does not get stuck in less than k steps.

$$\mathsf{Safe}_k = \{a \mid \forall j{<}k.\ a \to^j b \Rightarrow (b \in \mathsf{Val} \;\vee\; \exists b'.\ b \to b')\}$$

Lemma 3.1.10. *If $a :_k \tau$, then $a \in \mathsf{Safe}_k$*

Proof. Let $a :_k \tau$, $j < k$ and $a \to^j b$. We have to show that b is a value or it is reducible. Assume by contradiction that b is not a value and b is irreducible, so b is stuck. But because b is irreducible, by Def. 3.1.4 we get that $\langle k{-}j, b\rangle \in \tau$, which implies that b is a value. This is a contradiction, so b cannot be stuck and thus $a \in \mathsf{Safe}_k$. $\square$

Definition 3.1.11 (Safety)**.** We call a term *safe*, if it does not get stuck for any number of steps.

$$\mathsf{Safe} = \{a \mid \forall k{\geq}0.\ a \in \mathsf{Safe}_k\}$$

The purpose of a type system is to prevent terms that could get stuck from being executed, or conversely to guarantee that well-typed terms do not get stuck. In a step-indexed semantic model, the definition of the semantic typing judgement directly enforces the safety of well-typed terms.

Theorem 3.1.12 (Type Soundness)**.** *If $\emptyset \models a : \alpha$, then $a \in \mathsf{Safe}$*

Proof. Assume that $\emptyset \models a : \alpha$. By Def. 3.1.8 a is closed and for all substitutions σ we have that $\forall k{\geq}0.\ \sigma(a) :_k \alpha$. But since a is closed it will not be affected by σ and thus $\forall k{\geq}0.\ a :_k \alpha$. By Lemma 3.1.10 we get that $\forall k{\geq}0.\ a{\in}\mathsf{Safe}_k$, which by Def. 3.1.11 allows us to conclude that a is safe. $\square$

This result is much more direct than in a subject-reduction proof [WF94]. On the other hand, unlike with subject-reduction, the validity of the typing rules needs to be proved with respect to the model before they can be used for constructing valid type derivations. As previously showed by Amal Ahmed [Ahm04] this directly corresponds to the logical relations[1] proof technique [Tai67, Gir72].

[1]In a setting with dynamically allocated state it corresponds to Kripke logical relations [JT93, OR95, Mit96].

3.2 Method Types

Even though methods are similar to procedures, unlike procedures methods are not self-contained. There is a tight coupling between the methods of the same object, and features such as method extraction would be unsound in the presence of subtyping [AC96a]. For this reason in the ς-calculus methods are not values or terms, and thus the method types we define in this section are not semantic types in the strict interpretation of Def. 3.1.1. Nevertheless, we call them method types since they are closed under descending index. Treating method types separately simplifies to some extent the inherently complex definitions of object types.

Intuitively, a method has a method type $\alpha \rightsquigarrow \tau$ if when invoked with a self argument of type α it produces a result of type τ. Then object types are just collections of methods of appropriate method types. Such a direct definition of method and object types is circular: in order to define the type of an object we need the type of all its methods, and in order to define the type of a method we need the type of the whole object.

This inherent recursion can be made well-founded by approximating types using step-indexing. In order to show that a method has a method type $\alpha \rightsquigarrow \tau$ for k steps, we only need to know that the method works correctly when invoked with self arguments that have type α for $k-1$ steps. This holds because invoking the method takes one computation step. Additionally the method can be invoked after some computation steps, so the result of invoking the method with an appropriate argument must be in τ for every $j < k$ steps.

More precisely, $\alpha \rightsquigarrow \tau$ is the set of all pairs of the form $\langle k, \varsigma(x)b\rangle$ such that for all $j < k$ and for all values v that have type α with approximation j, the term obtained from substituting x with v in b has type τ with approximation j. This substitution directly corresponds to the operational semantics of method invocation (rule Red-Inv in Figure 2.1).

Definition 3.2.1 (Method Types)**.** If α and τ are semantic types, then

$$\alpha \rightsquigarrow \tau \triangleq \{\langle k, \varsigma(x)b\rangle \mid \forall j{<}k.\ \forall v{\in}\mathsf{CVal}.\ \langle j, v\rangle \in \alpha \Rightarrow [x \mapsto v]\,(b) :_j \tau\}$$

Even if they are not semantic types, method types are closed under descending index.

Property 3.2.2. *If α and τ are semantic types, $j \leq k$ and $\langle k, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau$, then also $\langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau$.*

Proof. Immediate from Def. 3.2.1. □

Method creation satisfies a property analogous to the typing rule for abstractions in the λ-calculus [Bar92] (rule Lam in Figure 1.2), which we later use to prove the object construction rule sound.

Lemma 3.2.3 (Method Creation)**.** *If $\Sigma[x \mapsto \alpha] \models b : \tau$, then for all $k \geq 0$ and for all $\sigma :_k \Sigma$ we have that $\langle k, \varsigma(x)(\sigma\,[x\!\uparrow]\,(b))\rangle \in \alpha \rightsquigarrow \tau$.*

Proof. Let $k \geq 0$, $\sigma :_k \Sigma$, $j < k$ and $v \in \mathsf{CVal}$ such that $\langle j, v\rangle \in \alpha$. By Def 3.2.1 it remains to be shown that $[x \mapsto v]\,(\sigma\,[x\!\uparrow]\,(b)) :_j \tau$, or alternatively that $\sigma[x \mapsto v](b) :_j \tau$. From $\sigma :_k \Sigma$ and $j < k$ we obtain that $\sigma :_j \Sigma$, which together with $v :_j \alpha$ gives us that $\sigma[x \mapsto v] :_k \Sigma[x \mapsto \alpha]$. From $\Sigma[x \mapsto \alpha] \models b : \tau$ we now obtain that $\sigma[x \mapsto v](b) :_j \tau$, thus by Def 3.2.1 $\langle k, \varsigma(x)(\sigma\,[x\!\uparrow]\,(b))\rangle \in \alpha \rightsquigarrow \tau$. □

3.3 Object Types

Objects are collections of methods that return results but do not have arguments other than self, so accordingly the type of an object associates to each method its return type. The type of an object $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$ can be written as $[m_d : \tau_d]_{d\in D}$, where each method m_d has type τ_d.

Example 3.3.1 (Shape Type)**.** For example the shape object in example 2.3.3 has the type: $[pos : [x : Nat, y : Nat], area : Nat]$. We can say that *pos* is a field of a nested object type with two fields x and y storing natural numbers, and *area* is a method returning a natural number as a result. However, the type of an object does not actually make a distinction between fields and methods, which corresponds with the fact that syntactically and operationally in the ς-calculus fields are just special kinds of methods.

3.3.1 The Simplest Definition

We start with a simple and intuitive definition of object types. If τ_d are types, then the object type $\alpha = [m_d : \tau_d]_{d\in D}$ is the set of all pairs formed from an index k and an object value $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$, where each method $\varsigma(x_d)b_d$ has method type $\alpha \leadsto \tau_d$ with approximation k. This definition of objects is well-founded since the definition of method types depends on values having type α only with approximation strictly smaller than k (Def. 3.2.1).

Definition 3.3.2 (Simple Object Types)**.**

$$[m_d : \tau_d]_{d\in D} \triangleq \{\langle k, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \mid \forall d{\in}D, \langle k, \varsigma(x_d)b_d\rangle{\in} [m_d : \tau_d]_{d\in D} \leadsto \tau_d\}$$

It turns out that this definition validates the usual typing rules for object types given in Figure 3.1.

According to rule Obj an object has an object type if the methods in the object are exactly the ones specified by the type and if we can independently give each of these methods their appropriate type, under the additional assumption that the self argument has the type of the whole object. As we will see in an example this turns out to be quite powerful.

A method invocation type checks when the target has an object type containing the invoked method. The type of the invocation is then the type of the invoked method (Inv).

The rule for method update (Upd) is similar, the method needs to already exist and have the same type as the updating one. Therefore method update preserves the type of the target object.

In the following we give some example type derivations using the rules.

Example 3.3.3 (Invocation)**.** Consider the term $[f{=}[], m{=}\varsigma(x)x.f].m$, which constructs an object with a field f storing the empty object and a method m returning the content of f, and then invokes m on this object. The type of this term is the empty object type[2] as illustrated by the following derivation. For

[2]By [] we denote both the empty object and the empty object type, however it should be clear from the context which one to consider.

Let $\alpha \equiv [m_d : \tau_d]_{d \in D}$

$$(\textsc{Obj})\ \frac{\forall d \in D.\ \Sigma[x_d \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha} \qquad (\textsc{Var})\ \Sigma \models x : \Sigma(x)$$

$$(\textsc{Inv})\ \frac{\Sigma \models a : \alpha \quad e \in D}{\Sigma \models a.m_e : \tau_e} \qquad (\textsc{Upd})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \Sigma[x \mapsto \alpha] \models b : \tau_e}{\Sigma \models a.m_e \coloneqq \varsigma(x)b : \alpha}$$

Figure 3.1: Typing rules for object creation, method invocation and update

brevity we denote $\Sigma \equiv [x \mapsto [f : [], m : []]]$

$$(\textsc{Inv})\ \frac{(\textsc{Obj})\ \dfrac{(\textsc{Obj})\ \Sigma \models [] : [] \qquad \dfrac{\Sigma \models x : [f : [], m : []]\ (\textsc{Var})}{\Sigma \models x.f : []}\ (\textsc{Obj})}{\emptyset \models [f{=}[], m{=}\varsigma(x)x.f] : [f : [], m : []]}}{\emptyset \models [f{=}[], m{=}\varsigma(x)x.f].m : []}$$

Example 3.3.4 (Showing Non-termination)**.** Only with object types we can already type the non-terminating program from Example 2.2.1. This illustrates the power of the object creation rule Obj, which allows us to add the premise that the self variable x has type $[m : \alpha]$ when checking that $x.m$ has type α.

$$(\textsc{Inv})\ \frac{(\textsc{Obj})\ \dfrac{(\textsc{Inv})\ \dfrac{(\textsc{Var})\ [x \mapsto [m : \alpha]] \models x : [m : \alpha]}{[x \mapsto [m : \alpha]] \models x.m : \alpha}}{\emptyset \models [m{=}\varsigma(x)x.m] : [m : \alpha]}}{\emptyset \models [m{=}\varsigma(x)x.m].m : \alpha}$$

Please note that in the derivation above α can be any type including $\bot$. Since there are no values of the empty type $\bot$, and since well-typed terms do not get stuck, we can conclude that the term is non-terminating. This example clearly illustrates that all types including $\bot$ are inhabited, and also that terms do not have unique types.

Example 3.3.5 (Update)**.** The following derivation illustrates that it is possible to update a method to a field in a type-safe way. The method is initialized to the diverging computation, so it has any type including the empty object type which we consider for this derivation.

$$(\textsc{Inv})\ \frac{(\textsc{Obj})\ \dfrac{(\textsc{Inv})\ \dfrac{(\textsc{Var})\ [x \mapsto [m : []]] \models x : [m : []]}{[x \mapsto [m : []]] \models x.m : []} \qquad \ldots \models [] : []\ (\textsc{Obj})}{\emptyset \models [m{=}\varsigma(x)x.m] : [m : []]}}{\emptyset \models [m{=}\varsigma(x)x.m].m \coloneqq [] : [m : []]}$$

Converting fields into methods is also possible.

3.3.2 Subtyping

"No object calculus can fully justify its existence without some notion of subsumption" – Luca Cardelli [AC96a]

The usual way to formalize subsumption is by defining a subtyping relation on types. A type α is a subtype of another type β (which for reasons that will become evident we write as $\alpha \subseteq \beta$), if all terms having type α also have type β. We express this as the following subsumption rule which allows us to use a term of a more specific[3] type than the context requires:

$$(\textsc{Sub})\ \frac{\Sigma \models a : \alpha \quad \alpha \subseteq \beta}{\Sigma \models a : \beta}$$

Since in a step-indexed model types are just sets, the natural subtyping relation is set inclusion. This subtyping relation forms a complete lattice on types. The least upper bound is given by the union and the greatest lower bound by the intersection. The least element is $\bot$ and the greatest is $\top$. The properties commonly used when type checking are given below as rules:

$$(\textsc{SubRefl})\ \alpha \subseteq \alpha \qquad (\textsc{SubTrans})\ \frac{\alpha \subseteq \tau \quad \tau \subseteq \beta}{\alpha \subseteq \beta}$$

$$(\textsc{SubTop})\ \alpha \subseteq \top \qquad (\textsc{SubBot})\ \bot \subseteq \alpha$$

The least we can expect from an object calculus in terms of subsumption is that an object with more methods is able to subsume any object with fewer methods of the same types. The intuition is that the object with more methods can emulate the one with fewer ones in every context, since the extra methods do not cause any harm. An object type should thus be a subtype of every object type containing only some of its methods. We call this *subtyping in width* and formalize it as the following rule:

$$(\textsc{SubObj})\ \frac{E \subseteq D}{[m_d : \tau_d]_{d \in D} \subseteq [m_e : \tau_e]_{e \in E}}$$

Unfortunately the simple definition of object types we considered so far does not validate this subtyping rule. There are two reasons for this, the first one is easy to understand and fix, while the second is more subtle.

The first reason subtyping for objects fails is that in Def. 3.3.2 we require an object type to contain only the objects which have *exactly* the methods defined by it. This clearly rules out all objects which have not only the methods of the type but also additional ones. An easy way to repair this is to allow objects with

[3]We commonly say that α is more specific than β, or β is more general than α, when α is a subtype of β.

any additional methods to appear in the object type. The candidate definition would be:

$$\begin{aligned}[m_d : \tau_d]_{d \in D} \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e \in E}\rangle \mid D \subseteq E,\\ \forall d{\in}D, \langle k, \varsigma(x_d)b_d\rangle \in [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_d\}\end{aligned}$$

Please notice the extra condition $D \subseteq E$, which allows the methods of the object to be a superset of the methods listed by its type.

While this is a good start, this definition still does not validate SubObj. In order to understand why we need to look at subtyping (i.e. set inclusion) between method types. The rule corresponds to the usual subtyping for function types in the λ-calculus [Bar92]. Intuitively a method of type $\alpha \rightsquigarrow \beta$ will also work for any self argument of a type more specific than α, and the result it produces can be viewed as having a more general type than β.

$$(\textsc{MethSub})\ \frac{\alpha' \subseteq \alpha \quad \beta \subseteq \beta'}{\alpha \rightsquigarrow \beta \subseteq \alpha' \rightsquigarrow \beta'}$$

We say that the method type constructor is contravariant in the type of the self argument and covariant in the result type. More formally in our model covariance corresponds to monotonicity, while contravariance corresponds to anti-monotonicity.

Definition 3.3.6 (Covariance)**.** A type constructor[4] F is *covariant* if

$$\forall \alpha, \beta {\in} \mathsf{Type}.\ \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq F(\beta)$$

Definition 3.3.7 (Contravariance)**.** A type constructor F is *contravariant* if

$$\forall \alpha, \beta {\in} \mathsf{Type}.\ \alpha \subseteq \beta \Rightarrow F(\beta) \subseteq F(\alpha)$$

Definition 3.3.8 (Invariance)**.** A type constructor is (strictly) *invariant* if it is neither covariant, nor contravariant.

The contravariance of method types in their argument type causes the following attempt to prove SubObj to fail.

Proof Attempt. Let $E \subseteq D$. We would like to show that $[m_d : \tau_d]_{d \in D} \subseteq [m_e : \tau_e]_{e \in E}$, or equivalently that for all $k \geq 0$ if $\langle k, v\rangle \in [m_d : \tau_d]_{d \in D}$ then also $\langle k, v\rangle \in [m_e : \tau_e]_{e \in E}$. We attempt to prove this by a course-of-values induction on k.

Let $k \geq 0$ and $\langle k, v\rangle \in [m_d : \tau_d]_{d \in D}$. By the definition above we have that $v = [m_{d'}{=}\varsigma(x_{d'})b_{d'}]_{d' \in D'}$ where $D \subseteq D'$ and for all $d \in D$ we have $\langle k, \varsigma(x_d)b_d\rangle \in [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_d$. By transitivity we have that $E \subseteq D'$. Let $e \in E$, since $E \subseteq D$ we have that $\langle k, \varsigma(x_e)b_e\rangle \in [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_e$. From this we would need to show that $\langle k, \varsigma(x_e)b_e\rangle \in [m_e : \tau_e]_{e \in E} \rightsquigarrow \tau_e$.

The induction hypothesis is that for all $j < k$ if $\langle k, v\rangle \in [m_d : \tau_d]_{d \in D}$ then also $\langle k, v\rangle \in [m_e : \tau_e]_{e \in E}$. By Lemma 3.4.12, which is just a more precise version of MethSub we give later, we obtain that if $\langle k, \varsigma(x)b\rangle \in [m_e : \tau_e]_{e \in E} \rightsquigarrow \tau_e$ then also $\langle k, \varsigma(x)b\rangle \in [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_e$. Please notice that this is the opposite of what we want to show, and the two statements are the same only for $E = D$, but in that case the conclusion is trivially true anyway.

[4] A type constructor is just a function from types to types.

So the contravariance of method types in their self argument causes SubObj to fail. The only solution we found to overcome this is to "unroll" the definition of method types and only require methods to work with the current object as the self argument.

$$\begin{aligned}[m_d : \tau_d]_{d \in D} \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e \in E}\rangle \mid\ & D \subseteq E,\\ & \forall d{\in}D, \forall j{<}k.\ ([x_d \mapsto v]\,(b_d) :_j \tau_d\}\end{aligned}$$

This definition turns objects into records of methods with proper subtyping, similar to the ones from the introduction. The method update rule Upd is clearly invalidated, however adding an extra condition finally solves this. We require that the object obtained by replacing a method with one of a proper type stays in the original object type, but only with a strictly smaller approximation index. Decreasing the index is once again crucial to ensure the well-foundedness of the definition.

Definition 3.3.9 (Object Types with Subtyping)**.** Let $\alpha \equiv [m_d : \tau_d]_{d \in D}$ in

$$\begin{aligned}\alpha \triangleq \{\langle k, v\rangle \mid\ & v \equiv [m_e{=}\varsigma(x_e)b_e]_{e \in E}, D \subseteq E,\\ & \forall d{\in}D.\ \forall j{<}k.\ ([x_d \mapsto v]\,(b_d) :_j \tau_d\\ & \quad \wedge\ \forall \varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d\\ & \qquad \Rightarrow \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e \in E \setminus \{d\}}\rangle \in \alpha)\}\end{aligned}$$

With this definition we can prove the soundness of the semantic rules from Fig. 3.1 as well as the soundness of the rule for subtyping in width (SubObj).

Example 3.3.10 (Minimal Types)**.** Unlike in a syntactic setting, the absence of type annotations does not cause the ς-calculus to lose the minimal types property. Consider for example the term $[m{=}\varsigma(x)[m{=}\varsigma(y)[]]]$ used by Abadi and Cardelli to show that the minimal types property breaks if type annotations are removed from their calculus [AC96a, Section 8.3.1]. As the following derivations show this term has both type $[m : [m : []]]$ and type $[m : []]$:

$$(\textsc{Obj})\ \frac{(\textsc{Obj})\ \dfrac{(\textsc{Obj})\ \ldots \models [] : []}{\ldots \models [m{=}\varsigma(y)[]] : [m : []]}}{\emptyset \models [m{=}\varsigma(x)[m{=}\varsigma(y)[]]] : [m : [m : []]]}$$

$$(\textsc{Obj})\ \frac{(\textsc{Sub})\ \dfrac{(\textsc{Obj})\ \dfrac{(\textsc{Obj})\ \ldots \models [] : []}{\ldots \models [m{=}\varsigma(y)[]] : [m : []]} \qquad \dfrac{\emptyset \subseteq \{m\}}{[m : []] \subseteq []}\ (\textsc{SubObj})}{\ldots \models [m{=}\varsigma(y)[]] : []}}{\emptyset \models [m{=}\varsigma(x)[m{=}\varsigma(y)[]]] : [m : []]}$$

In the type system of Abadi and Cardelli these types would have no common subtype. However, in our semantic model every collection of types has a common subtype given by their intersection. In particular $[m : [m : []]]$ and $[m : []]$ have $[m : [m : []]] \cap [m : []]$ as their most general common subtype (i.e. greatest lower bound), and our term can also be given this intersection type [Pie92, BDCd95].

$$\frac{\emptyset \models [m{=}\varsigma(x)[m{=}\varsigma(y)[]]] : [m : [m : []]] \qquad \emptyset \models [m{=}\varsigma(x)[m{=}\varsigma(y)[]]] : [m : []]}{\emptyset \models [m{=}\varsigma(x)[m{=}\varsigma(y)[]]] : [m : [m : []]] \cap [m : []]}$$

Subtyping in width considers objects types with more methods to be subtypes of objects with less methods, as long as the type of the common methods is exactly the same. Under this assumption we say that object types are invariant in their components. We may wonder if we can relax this to allow the components to be subtyped as well. For example records satisfy the following rule for covariant *subtyping in depth*[5]:

$$\dfrac{\forall d{\in}D.\ \alpha_d \subseteq \beta_d}{\{f_d : \alpha_d\}_{d\in D} \subseteq \{f_d : \beta_d\}_{d\in D}}$$

This rule holds for records only because they are immutable. However method update changes the state of objects, so subtyping in depth would be unsound in any model as the following counterexample shows.

Counterexample 3.3.11 (Subtyping in Depth)**.** Please consider the following term $([f{=}[g{=}[]], m{=}\varsigma(x)x.f.g].f \coloneqq []).m$. It creates an object with a field f and a method m, it updates the field f to the empty object and then it invokes the method m. The field f initially stores an object with one field g containing the empty object, and the method selects the content of this inner field, so the type of the initially constructed object is $[f : [g : []], m : []]$ as shown by the derivation t below (we denote $\Sigma \equiv [x \mapsto [f : [g : []], m : []]]$ and $\Sigma' \equiv \Sigma[\_ \mapsto [g : []]]$).

$$t \;=\; (\textsc{Obj})\ \dfrac{(\textsc{Obj})\ \dfrac{(\textsc{Obj})\ \Sigma' \models [] : []}{\Sigma \models [g{=}[]] : [g : []]} \qquad \dfrac{\dfrac{(\textsc{Var})\ \Sigma \models x : [f : [g : []], m : []]}{\Sigma \models x.f : [g : []]}\,(\textsc{Inv})}{\Sigma \models x.f.g : []}\,(\textsc{Inv})}{\emptyset \models [f{=}[g{=}[]], m{=}\varsigma(x)x.f.g] : [f : [g : []], m : []]}$$

We cannot directly apply the Upd rule since the update of f is not type preserving. However, if we had a rule for covariant subtyping in depth we could derive that $[f : [g : []], m : []] \subseteq [f : [], m : []]$, so by subsumption the object that we initially constructed also has type $[f : [], m : []]$. Now the update would be permitted and the whole term would be considered well-typed.

$$(\textsc{Obj})\ \dfrac{(\textsc{Upd})\ \dfrac{(\textsc{Sub})\ \dfrac{t \qquad \dfrac{[g : []] \subseteq []}{[f : [g : []], m : []] \subseteq [f : [], m : []]}\,(\textsc{Wrong!})}{\emptyset \models [f{=}[g{=}[]], m{=}\varsigma(x)x.f.g] : [f : [], m : []]}}{\emptyset \models [f{=}[g{=}[]], m{=}\varsigma(x)x.f.g].f \coloneqq [] : [f : [], m : []]}}{\emptyset \models ([f{=}[g{=}[]], m{=}\varsigma(x)x.f.g].f \coloneqq []).m : []}$$

However, it is easy to see that this term gets stuck in three steps because the method m tries to access the inner field g which no longer exists after the update.

$$\begin{aligned}([f{=}[g{=}[]], m{=}\varsigma(x)x.f.g].f \coloneqq []).m &\to [f{=}[], m{=}\varsigma(x)x.f.g].m \\ &\to [f{=}[], m{=}\varsigma(x)x.f.g].f.g \to [].g \not\to\end{aligned}$$

If contravariant subtyping in depth is allowed then we can construct a similar counterexample. Intuitively, methods can be both invoked (read) and updated (written) so like the references in ML [MTM96] they need to be invariant when

[5]For simplicity we did not also consider subtyping in width in the rule.

subtyping[6]. However, subtyping in depth is important because it would increase the expressive power of the type system. For example with the encoding of procedures from Section 2.3.1, the type of procedures $\alpha \to \beta$ can be encoded as $[\mathit{arg} : \alpha, \mathit{eval} : \beta]$. However, with object types that are invariant in their components such an encoding clearly does not validate the usual subtyping rule for procedures:

$$\frac{\alpha' \subseteq \alpha \quad \beta \subseteq \beta'}{\alpha \to \beta \subseteq \alpha' \to \beta'} \;\Leftrightarrow\; \frac{\alpha' \subseteq \alpha \quad \beta \subseteq \beta'}{[\mathit{arg} : \alpha, \mathit{eval} : \beta] \subseteq [\mathit{arg} : \alpha', \mathit{eval} : \beta']}\ (\textsc{Wrong!})$$

In the next section we show that by tracking the variance of the methods and restricting certain invocations or updates we can still soundly allow contravariant respectively covariant subtyping in depth for object types.

Subtyping in a step-indexed semantic model was previously considered by Swadi [Swa03, Section 3.4]. However, his typed machine language is much different than the ς-calculus. In particular it does not have object types, so many of the subtle issues discussed in this section do not appear there.

3.4 Object Types with Variance Annotations

3.4.1 Definition and Rules

Abadi and Cardelli discovered that, by restricting certain updates and invocations, covariant respectively contravariant subtyping in depth for object types can be soundly allowed [AC96a, Section 8.7]. Each method of an object type $[m_d :_{\nu_d} \tau_d]_{d \in D}$ is assigned a variance annotation $\nu \in \{0, +, -\}$.

Methods annotated with "0" correspond to the ones we have considered so far. They can be both invoked and updated, but as before they need to be considered *invariant* when subtyping. Methods annotated with "+" can only be invoked but not updated, so they are treated as *covariant* when subtyping. Similarly, methods marked with "−" can only be updated, and are thus *contravariant.*

The semantic typing rules for objects are slightly changed (Figure 3.2). The rule for object creation is the same as before, just that we can also chose the variance annotations in the type of the created object (VarObj). A method invocation is well-typed only if the method is marked as covariant or invariant (VarInv). On the other hand, a method update is allowed only if the method is contravariant or invariant (VarUpd). The new rule for subtyping allows not only subtyping in width, but also in depth (VarSubObj1). As expected an object type is covariant in its methods annotated with +, contravariant in the ones annotated with −, and invariant in the ones annotated with 0 (if ν_e is 0 then we assume both $\alpha_e \subseteq \beta_e$ and $\beta_e \subseteq \alpha_e$, which is just $\alpha_e = \beta_e$).

If we consider only these typing rules the type system is not more flexible than the one we had before. Methods marked as covariant are effectively read-only, like the fields of a record. Contravariant methods are write-only "black holes", which can never be read so they hardly have any use. VarSubObj2 is the typing rule that adds the missing flexibility.

[6]For the same reason Java arrays should have been invariant, but due to a poor design choice they are covariant. This is unsound and is rectified by doing costly dynamic checks on each array write operation.

Let $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d\in D}$

$$(\textsc{VarObj})\ \frac{\forall d \in D.\ \Sigma[x \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha}$$

$$(\textsc{VarInv})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \nu_e \in \{+, 0\}}{\Sigma \models a.m_e : \tau_e}$$

$$(\textsc{VarUpd})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \nu_e \in \{-, 0\} \quad \Sigma[x \mapsto \alpha] \models b : \tau_e}{\Sigma \models a.m_e \coloneqq \varsigma(x)b : \alpha}$$

$$(\textsc{VarSubObj1})\ \frac{\begin{array}{r} E \subseteq D \quad \forall e \in E.\ (\nu_e \in \{+, 0\} \Rightarrow \alpha_e \subseteq \beta_e) \\ \wedge\ (\nu_e \in \{-, 0\} \Rightarrow \beta_e \subseteq \alpha_e) \end{array}}{[m_d :_{\nu_d} \alpha_d]_{d\in D} \subseteq [m_e :_{\nu_e} \beta_e]_{e\in E}}$$

$$(\textsc{VarSubObj2})\ \frac{\forall d \in D.\ \nu_d = 0\ \vee\ \nu_d = \nu'_d}{[m_d :_{\nu_d} \tau_d]_{d\in D} \subseteq [m_d :_{\nu'_d} \tau_d]_{d\in D}}$$

Figure 3.2: Typing rules for the object types with variance annotations

VarSubObj2 allows invariant methods to be regarded by subtyping as either covariant or contravariant. Intuitively, this is sound since a method that can be both invoked and updated can be safely used in a context where it is only invoked, or in one where it is only updated. VarSubObj2 gives us for example that $[m :_0 \alpha]$ is a subtype of both $[m :_+ \alpha]$ and $[m :_- \alpha]$, while $[m :_+ \alpha]$ and $[m :_- \alpha]$ are not comparable to each other. This allows us to distinguish between the invocations and updates done through the self argument (internal), and the ones done from the outside (external).

The main idea is to type an object creation with an object type where all methods are considered invariant, so that all invocations and updates through the self argument (internal) are allowed, but have to be type preserving. Then the subsumption rule is applied and some of the methods can become covariant, some of them contravariant. Any external update to a covariant method and any external invocation of a contravariant one are prohibited. However, the internal invocation and update remain unrestricted. This is later illustrated in Section 3.4.2.

The new definition of object types follows naturally from the previous one (Def. 3.3.9), which has a part about invocations, and a separate part about updates. As before invariant methods have to allow both invocations and updates. Covariant methods have to allow invocations, but not necessarily updates, while contravariant ones have to allow updates, but not necessarily invocations.

Definition 3.4.1 (Object Types with Variance Annotations)**.** $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d\in D}$

$$\begin{aligned}\alpha \triangleq \{\langle k, v\rangle \mid v \equiv [m_e{=}\varsigma(x_e)b_e]_{e\in E}\,, D \subseteq E,\\ \forall d{\in}D.\ \forall j{<}k.\ ((\nu_d \in \{+,0\} \Rightarrow [x_d \mapsto v]\,(b_d) :_j \tau_d)\\ \wedge\ (\nu_d \in \{-,0\} \Rightarrow \forall\varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d\\ \Rightarrow \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha))\}\end{aligned}$$

The simple object types considered in the previous section can be seen as a special case of object types with variance annotations where all methods are invariant.

3.4.2 First-order Encoding of Procedure Types

Object types with variance annotations are more expressive than the simpler ones without. With the encoding of procedures from Section 2.3.1 we can give an encoding of procedure types that validates the usual subtyping rule, since $\langle\!\langle\alpha \to \beta\rangle\!\rangle$ can be defined as $[\mathit{arg} :_- \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_+ \langle\!\langle\beta\rangle\!\rangle]$:

$$(\textsc{VarSubObj1})\ \frac{\langle\!\langle\alpha'\rangle\!\rangle \subseteq \langle\!\langle\alpha\rangle\!\rangle \qquad \langle\!\langle\beta\rangle\!\rangle \subseteq \langle\!\langle\beta'\rangle\!\rangle}{[\mathit{arg} :_- \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_+ \langle\!\langle\beta\rangle\!\rangle] \subseteq [\mathit{arg} :_- \langle\!\langle\alpha'\rangle\!\rangle\,, \mathit{eval} :_+ \langle\!\langle\beta'\rangle\!\rangle]}$$

$$\Downarrow$$

$$(\textsc{SubProc})\ \frac{\langle\!\langle\alpha'\rangle\!\rangle \subseteq \langle\!\langle\alpha\rangle\!\rangle \quad \langle\!\langle\beta\rangle\!\rangle \subseteq \langle\!\langle\beta'\rangle\!\rangle}{\langle\!\langle\alpha \to \beta\rangle\!\rangle \subseteq \langle\!\langle\alpha' \to \beta'\rangle\!\rangle}$$

We first give any encoded procedure the type $[\mathit{arg} :_0 \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_0 \langle\!\langle\beta\rangle\!\rangle]$, so that the argument field can be selected and used through the self reference. Then the subsumption rule is applied and the object is given type $[\mathit{arg} :_- \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_+ \langle\!\langle\beta\rangle\!\rangle]$, which by VarSubObj2 is a supertype of $[\mathit{arg} :_0 \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_0 \langle\!\langle\beta\rangle\!\rangle]$. External selections of the *arg* field and external updates of *eval* are not allowed by any sound typing rules. However this does not impede internal access, and is in concordance with the intended usage pattern of an encoded procedure. An encoded application first updates the field to the actual argument, and then invokes the *eval* method that accesses it through self.

Example 3.4.2 (Eta)**.** As an example we show that:

$$[f \mapsto \langle\!\langle\alpha \to \beta\rangle\!\rangle] \models \langle\!\langle\lambda x.\, f\ x\rangle\!\rangle : \langle\!\langle\alpha \to \beta\rangle\!\rangle$$

We use the following notations:

$$\begin{aligned} a &\equiv\ \langle\!\langle\lambda x.\, f\ x\rangle\!\rangle = [\mathit{arg}{=}\varsigma(x)x.\mathit{arg}, \mathit{eval}{=}\varsigma(x)(f.\mathit{arg} := x.\mathit{arg}).\mathit{eval}]\\ \tau &\equiv\ \langle\!\langle\alpha \to \beta\rangle\!\rangle = [\mathit{arg} :_- \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_+ \langle\!\langle\beta\rangle\!\rangle]\\ \tau_0 &\equiv [\mathit{arg} :_0 \langle\!\langle\alpha\rangle\!\rangle\,, \mathit{eval} :_0 \langle\!\langle\beta\rangle\!\rangle]\\ \Sigma &\equiv\ [f \mapsto \tau] \qquad \Sigma' \equiv \Sigma[x \mapsto \tau_0]\end{aligned}$$

Please note that a cannot be given type τ directly, since it selects the field *arg* which is contravariant in τ. However, it can be given type τ_0 that is the same as τ just that all methods are invariant, and thus by subsumption a can still be given type τ. The complete typing derivation is given below.

$$
(\textsc{Sub})\ \dfrac{(\textsc{VarObj})\ \dfrac{(\textsc{VarInv})\ \dfrac{(\textsc{Var})\ \Sigma' \models x : \tau_0}{\Sigma' \models x.arg : \langle\!\langle \alpha \rangle\!\rangle} \qquad \dfrac{\dfrac{(\textsc{Var})\ \Sigma' \models f : \tau \qquad \dfrac{\Sigma' \models x : \tau_0\ (\textsc{Var})}{\Sigma' \models x.arg : \langle\!\langle \alpha \rangle\!\rangle}\ (\textsc{VarInv})}{\Sigma' \models f.arg := x.arg : \tau}\ (\textsc{VarUpd})}{\Sigma' \models (f.arg := x.arg).eval : \langle\!\langle \beta \rangle\!\rangle}\ (\textsc{VarInv})}{\Sigma \models a : \tau_0} \qquad (\textsc{VarSubObj2})\ \tau_0 \subseteq \tau}{\Sigma \models a : \tau}
$$

The semantic typing rules for application and abstraction can be proved from the rules for objects. We start with a simple result about substitution.

Lemma 3.4.3 (Substitution)**.** *If* $\Sigma[x \mapsto \alpha] \models b : \beta$ *and* $\Sigma \models a : \alpha$*, then* $\Sigma \models [x \mapsto a]\,(b) : \beta$

Proof. Let $k \geq 0$, $\sigma :_k \Sigma$. From $\Sigma \models a : \alpha$ by Def. 3.1.8 we have that $\sigma(a) :_k \alpha$, so also $\sigma[x \mapsto a] :_k \Sigma[x \mapsto \alpha]$. From $\Sigma \models a : \alpha$ we have that $\sigma[x \mapsto a](b) :_k \beta$, so finally by Def. 3.1.8 we can conclude that $\Sigma \models [x \mapsto a]\,(b) : \beta$. $\square$

Lemma 3.4.4 (Lam: Abstraction)**.**
If $\Sigma[x \mapsto \langle\!\langle \alpha \rangle\!\rangle] \models \langle\!\langle b \rangle\!\rangle : \langle\!\langle \beta \rangle\!\rangle$ *then* $\Sigma \models \langle\!\langle \lambda x.\, b \rangle\!\rangle : \langle\!\langle \alpha \to \beta \rangle\!\rangle$.

Proof. We make the following notations:

$$
\begin{aligned}
a &\equiv \langle\!\langle \lambda x.\, b \rangle\!\rangle = [arg{=}\varsigma(x)x.arg,\ eval{=}\varsigma(x)\,[x \mapsto x.arg]\,(\langle\!\langle b \rangle\!\rangle)] \\
\tau &\equiv \langle\!\langle \alpha \to \beta \rangle\!\rangle = [arg :_{-} \langle\!\langle \alpha \rangle\!\rangle\,,\ eval :_{+} \langle\!\langle \beta \rangle\!\rangle] \\
\tau_0 &\equiv [arg :_0 \langle\!\langle \alpha \rangle\!\rangle\,,\ eval :_0 \langle\!\langle \beta \rangle\!\rangle] \\
\Sigma' &\equiv \Sigma[x \mapsto \tau_0]
\end{aligned}
$$

The derivation is similar to the one in example 3.4.2.

$$
(\textsc{Sub})\ \dfrac{(\textsc{VarObj})\ \dfrac{(\textsc{VarInv})\ \dfrac{(\textsc{Var})\ \Sigma' \models x : \tau_0}{\Sigma' \models x.arg : \langle\!\langle \alpha \rangle\!\rangle} \qquad t}{\Sigma \models a : \tau_0} \qquad \tau_0 \subseteq \tau\ (\textsc{VarSubObj2})}{\Sigma \models a : \tau}
$$

$$
\text{where } t = \dfrac{(\textsc{VarInv})\ \dfrac{(\textsc{Var})\ \Sigma' \models x : \tau_0}{\Sigma' \models x.arg : \langle\!\langle \alpha \rangle\!\rangle} \qquad \Sigma'[x \mapsto \langle\!\langle \alpha \rangle\!\rangle] \models \langle\!\langle b \rangle\!\rangle : \langle\!\langle \beta \rangle\!\rangle}{\Sigma' \models [x \mapsto x.arg]\,(\langle\!\langle b \rangle\!\rangle) : \langle\!\langle \beta \rangle\!\rangle}\ (\textsc{Subst})
$$

$\square$

Lemma 3.4.5 (App: Application)**.** *If* $\Sigma \models \langle\!\langle a \rangle\!\rangle : \langle\!\langle \beta \to \alpha \rangle\!\rangle$ *and* $\Sigma \models \langle\!\langle b \rangle\!\rangle : \langle\!\langle \beta \rangle\!\rangle$*, then* $\Sigma \models \langle\!\langle a\ b \rangle\!\rangle : \langle\!\langle \alpha \rangle\!\rangle$.

Proof. Let $\tau \equiv \langle\!\langle \beta \to \alpha \rangle\!\rangle = [arg :_{-} \langle\!\langle \alpha \rangle\!\rangle\,,\ eval :_{+} \langle\!\langle \beta \rangle\!\rangle]$, then we have:

$$
(\textsc{Inv})\ \dfrac{(\textsc{Upd})\ \dfrac{\Sigma \models \langle\!\langle a \rangle\!\rangle : \tau \qquad \Sigma \models \langle\!\langle b \rangle\!\rangle : \langle\!\langle \beta \rangle\!\rangle}{\Sigma \models \langle\!\langle a \rangle\!\rangle .arg := \langle\!\langle b \rangle\!\rangle : \tau}}{\Sigma \models (\langle\!\langle a \rangle\!\rangle .arg := \langle\!\langle b \rangle\!\rangle).eval : \langle\!\langle \alpha \rangle\!\rangle}
$$

$\square$

In the following we omit the notation for encoded procedures and procedure types, and use procedures and procedure types as if they were primitive.

Procedure types with proper subtyping can also be properly encoded using the quantified types introduced in Section 3.6.

3.4.3 Soundness of the Semantic Typing Rules

We have delayed proving any of the semantic typing rules for objects so that we can give the proofs only in the more general setting of object types with variance annotations. The semantic typing rules for the object types without variance annotations from Section 3.3.2 are just corollaries.

Lemma 3.4.6. *If all τ_d are types, then $[m_d :_{\nu_d} \tau_d]_{d\in D}$ is also a type.*

Proof. The closure under descending index is immediate from Def. 3.4.1. □

Lemma 3.4.7 (Object Construction)**.** *For all $k \geq 0$, for all sets D, for all object types $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$, for all objects $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$, if for all $d \in D$ we have $\langle k, \varsigma(x_d)b_d\rangle \in \alpha \rightsquigarrow \tau_d$, then we can conclude that $\langle k, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \in \alpha$.*

Proof. By induction on k. Since the base case $k = 0$ is immediate from Def. 3.4.1 we only consider $k > 0$. Let D be an arbitrary set, let $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$ be an object type, and let $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$ be an object such that $\forall d \in D.\ \langle k, \varsigma(x_d)b_d\rangle \in \alpha \rightsquigarrow \tau_d$, we need to show that $\langle k, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \in \alpha$.

Let $j < k$. By closure under descending index (Property 3.2.2) we get that $\forall d \in D.\ \langle j, \varsigma(x_d)b_d\rangle \in \alpha \rightsquigarrow \tau_d$, so by the induction hypothesis we have that $\langle j, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \in \alpha$. Let $d \in D$. We already have that $\langle j, \varsigma(x_d)b_d\rangle \in \alpha \rightsquigarrow \tau_d$, thus by Def. 3.2.1 we obtain that $\left[x_d \mapsto [m_d{=}\varsigma(x_d)b_d]_{d\in D}\right](b_d) :_j \tau_d$, so $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$ satisfies the method invocation condition in Def. 3.4.1 indifferent to the variance annotations ν_d.

Let $\varsigma(x)b$ be a method such that $\langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d$. The updated object $[m_d{=}\varsigma(x)b, m_{d'}{=}\varsigma(x_{d'})b_{d'}]_{d'\in D\setminus\{d\}}$ satisfies the premises of the lemma, so by induction hypothesis we get that $\langle j, [m_d{=}\varsigma(x)b, m_{d'}{=}\varsigma(x_{d'})b_{d'}]_{d'\in D\setminus\{d\}}\rangle \in \alpha$, once again for any value of ν_d. Finally, by applying Def. 3.4.1 for $E{=}D$ we conclude that $\langle k, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \in \alpha$. □

Lemma 3.4.8 (VarObj: Object Construction)**.** *For all object types with variance annotations $\alpha = [m_d :_{\nu_d} \tau_d]_{d\in D}$, if for all $d \in D$ we have $\Sigma[x_d \mapsto \alpha] \models b_d : \tau_d$, then $\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$.*

Proof. Let $k \geq 0$ and $\sigma :_k \Sigma$. If we denote $v \equiv [m_d{=}\varsigma(x_d)(\sigma\,[x_d\!\uparrow]\,(b_d))]_{d\in D}$, then we must prove that $v :_k \alpha$. From the premises, by applying Lemma 3.2.3 for all $d \in D$ we obtain that $\langle k, \varsigma(x_d)(\sigma\,[x_d\!\uparrow]\,(b_d))\rangle \in \alpha \rightsquigarrow \tau$. Then by Lemma 3.4.7 we obtain that $\langle k, v\rangle \in \alpha$, and by Property 3.1.6 it follows that $v :_k \alpha$. Finally, by Def. 3.1.8 we conclude that $\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d\in D} : \alpha$. □

Note. The induction would fail unless we strengthen the hypothesis to the one in Lemma 3.4.7. The universal quantification over all objects with well-typed methods is crucial for proving that method update preserves the type α.

Lemma 3.4.9 (VarInv: Method Invocation)**.** *For all object types with variance annotations $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d \in D}$ and for all $e \in D$, if $\Sigma \models a : \alpha$ and $\nu_e \in \{+, 0\}$, then $\Sigma \models a.m_e : \tau_e$.*

Proof. Let $k > 0$ and $\sigma :_k \Sigma$. From the hypothesis by Def. 3.1.8 we get that $\sigma(a) :_k \alpha$. Also by Def. 3.1.8 it suffices to show that $\sigma(a).m_e :_k \tau_e$.

Let $j < k$ such that $\sigma(a).m_e \rightarrow^j b$ and $b \not\rightarrow$. Since $\sigma(a) :_k \alpha$ by Lemma 3.1.10 we get that $\sigma(a)$ is safe for k steps, so by the operational semantics we can infer that $\exists i < j$ such that $\sigma(a) \rightarrow^i v$ and v is a value thus irreducible. Since $\sigma(a) :_k \alpha$ by Def. 3.1.4 we get that $\langle k - i, v\rangle \in \alpha$. By the definition of object types (Def. 3.4.1) it follows that $v = [m_e{=}\varsigma(x_e)b_e]_{e \in E}$ where $D \subseteq E$, and since $\nu_e \in \{+, 0\}$ also that $[x_e \mapsto v]\,(b_e) :_{k-i-1} \tau_e$. From $\sigma(a) \rightarrow^i v$ by the rules Red-Ctx and Red-Inv it follows that $\sigma(a).m_e \rightarrow^i v.m_e \rightarrow [x_e \mapsto v]\,(b_e) \rightarrow^{j-i-1} b$. Therefore by Def. 3.1.4 we have that $\langle k - j, b\rangle \in \tau_e$, so again by Def. 3.1.4 we can conclude that $\sigma(a).m_e :_k \tau_e$. □

Lemma 3.4.10 (Closed Method Update)**.** *For all object types with variance annotations $\alpha = [m_d :_{\nu_d} \tau_d]_{d \in D}$, for all $d \in D$, for all closed terms a', and for all closed methods $\varsigma(x)b'$, if $a' :_k \alpha$ and $\langle k, \varsigma(x)b'\rangle \in \alpha \rightsquigarrow \tau_d$ and $\nu_d \in \{-, 0\}$, then $a'.m_d \coloneqq \varsigma(x)b' :_k \alpha$.*

Proof. Let $j < k$ such that $a'.m_d \coloneqq \varsigma(x)b' \rightarrow^j b''$ and $b'' \not\rightarrow$. From $a' :_k \alpha$ by Lemma 3.1.10 we get that $a' \in \mathsf{Safe}_k$, so by the operational semantics we have that $a'.m_d \coloneqq \varsigma(x)b' \rightarrow^{j-1} v.m_d \coloneqq \varsigma(x)b'$. From Red-Ctx by inversion we get that $a' \rightarrow^{j-1} v$. From $a' :_k \alpha$, by Def. 3.1.4 we obtain that $\langle k{-}j{+}1, v\rangle \in \alpha$. From the definition of α (Def. 3.4.1) we infer that $v = [m_e{=}\varsigma(x_e)b_e]_{e \in E}$ where $D \subseteq E$. From the rule Red-Upd we derive that $b'' = [m_d{=}\varsigma(x)b', m_e{=}\varsigma(x_e)b_e]_{e \in E \setminus \{d\}}$ and also $a'.m_d \coloneqq \varsigma(x)b' \rightarrow^{j-1} v.m_d \coloneqq \varsigma(x)b' \rightarrow b''$.

We have that $k{-}j < k{-}j{-}1$, and from the hypothesis that $\nu_d \in \{-, 0\}$ and $\langle k, \varsigma(x)b'\rangle \in \alpha \rightsquigarrow \tau_d$ so by Prop. 3.2.2 also $\langle k{-}j, \varsigma(x)b'\rangle \in \alpha \rightsquigarrow \tau_d$. From all these by Def. 3.4.1 we obtain that $\langle k{-}j, b''\rangle \in \alpha$. This together with $a'.m_d \coloneqq \varsigma(x)b' \rightarrow^j b''$ and $b'' \not\rightarrow$ allows us to conclude by Def. 3.1.4 that $a'.m_d \coloneqq \varsigma(x)b' :_k \alpha$. □

Lemma 3.4.11 (VarUpd: Method Update)**.** *For all object types with variance annotations $\alpha = [m_d :_{\nu_d} \tau_d]_{d \in D}$ and for all $e \in D$, if $\Sigma \models a : \alpha$ and $\Sigma[x \mapsto \alpha] \models b : \tau_e$ and $\nu_e \in \{-, 0\}$, then $\Sigma \models a.m_e \coloneqq \varsigma(x)b : \alpha$.*

Proof. Under the premises of the lemma we need to show that $\Sigma \models a.m_e \coloneqq \varsigma(x)b : \alpha$. Let $k \geq 0$ and $\sigma :_k \Sigma$, by Def. 3.1.8 it remains to be showed that $\sigma(a).m_e \coloneqq \varsigma(x)\sigma(b) :_k \alpha$. From the hypothesis $\Sigma \models a : \alpha$ by Def. 3.1.8 we obtain that $\sigma(a) :_k \alpha$. From the hypothesis $\Sigma[x \mapsto \alpha] \models b : \tau_e$ by Lemma 3.2.3 we also obtain that $\langle k, \varsigma(x)\sigma\,[x\!\uparrow]\,(b)\rangle \in \alpha \rightsquigarrow \tau_e$. By putting them all together and applying Lemma 3.4.10, we conclude that $\sigma(a).m_e \coloneqq \varsigma(x)\sigma(b) :_k \alpha$. □

Lemma 3.4.12 (Subtyping Method Types)**.** *If $\langle k, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau$, $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and $\lfloor \tau \rfloor_k \subseteq \lfloor \tau' \rfloor_k$, then $\langle k, \varsigma(x)b\rangle \in \alpha' \rightsquigarrow \tau'$.*

Proof. Assume that $\langle k, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau$, $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ and $\lfloor \tau \rfloor_k \subseteq \lfloor \tau' \rfloor_k$. We need to show that $\langle k, \varsigma(x)b\rangle \in \alpha' \rightsquigarrow \tau'$. Let $j < k$ and let v be a value such that $\langle j, v\rangle \in \alpha'$, or equivalently (by Def. 3.1.2) $\langle j, v\rangle \in \lfloor \alpha' \rfloor_k$. But by our assumption $\lfloor \alpha' \rfloor_k \subseteq \lfloor \alpha \rfloor_k$ so $\langle j, v \rangle \in \lfloor \alpha \rfloor_k$, and equivalently $\langle j, v \rangle \in \alpha$. From the assumption that $\langle k, \varsigma(x)b \rangle \in \alpha \rightsquigarrow \tau$ by Def. 3.2.1 we infer that $[x \mapsto v]\,(b) :_j \tau$, so $[x \mapsto v]\,(b) :_j \lfloor \tau \rfloor_k$. By our third assumption $\lfloor \tau \rfloor_k \subseteq \lfloor \tau' \rfloor_k$, thus by Def. 3.1.4 we have that $[x \mapsto v]\,(b) :_j \lfloor \tau' \rfloor_k$, which is again equivalent to $[x \mapsto v]\,(b) :_j \tau'$. Finally, by the definition of method types (Def. 3.2.1) we conclude that $\langle k, \varsigma(x)b \rangle \in \alpha' \rightsquigarrow \tau'$. □

Corollary 3.4.13 (MethSub: Subtyping Method Types). *If $\alpha' \subseteq \alpha$ and $\tau \subseteq \tau'$, then $\alpha \rightsquigarrow \tau \subseteq \alpha' \rightsquigarrow \tau'$.*

Proof. Immediate from Lemma 3.4.12. □

Lemma 3.4.14 (VarSubObj1: Subtyping Object Types). *$E \subseteq D$ and for all $e \in E$ if $\nu_e \in \{+, 0\}$ then $\alpha_e \subseteq \beta_e$ and if $\nu_e \in \{-, 0\}$ then $\beta_e \subseteq \alpha_e$ imply that $[m_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [m_e :_{\nu_e} \beta_e]_{e \in E}$.*

Proof. Assume that $E \subseteq D$ and that if $\nu_e \in \{+, 0\}$ then $\alpha_e \subseteq \beta_e$ and if $\nu_e \in \{-, 0\}$ then $\beta_e \subseteq \alpha_e$ (H). Let us denote $\alpha \equiv [m_d :_{\nu_d} \alpha_d]_{d \in D}$ and $\beta \equiv [m_e :_{\nu_e} \beta_e]_{e \in E}$. We prove that for all $k \geq 0$ and for all values v, if $\langle k, v \rangle \in \alpha$ then $\langle k, v \rangle \in \beta$, by course-of-values induction on k. The induction hypothesis is thus that for all $j < k$ if $\langle j, v \rangle \in \alpha$ then $\langle j, v \rangle \in \beta$, or equivalently $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$.

Assume that $\langle k, v \rangle \in \alpha$, then by the definition of α (Def. 3.4.1) we have that $v = [m_c{=}\varsigma(x_c)b_c]_{c \in C}$, $D \subseteq C$ and for all $d \in D$ and $j < k$ we have

$$\nu_d \in \{+, 0\} \Rightarrow [x_d \mapsto v]\,(b_d) :_j \alpha_d \tag{H1}$$

$$\begin{aligned}\nu_d \in \{-, 0\} \Rightarrow \forall \varsigma(x)b.\ \langle j, \varsigma(x)b \rangle \in \alpha \rightsquigarrow \alpha_d \Rightarrow \\ \langle j, [m_d{=}\varsigma(x)b, m_c{=}\varsigma(x_c)b_c]_{c \in C \setminus \{d\}} \rangle \in \alpha\end{aligned} \tag{H2}$$

Since $E \subseteq D$ and $D \subseteq C$ then by transitivity $E \subseteq C$. Let $e \in E$ and $j < k$. We first show that if $\nu'_e \in \{+, 0\}$ then $[x_e \mapsto v]\,(b_e) :_j \beta_e$. Assume that $\nu'_e \in \{+, 0\}$, so by the hypothesis (H) we have $\alpha_e \subseteq \beta_e$. By $E \subseteq D$ and (H1) it follows that $[x_e \mapsto v]\,(b_e) :_j \alpha_e$, so by Def. 3.1.4 it follows that $[x_e \mapsto v]\,(b_e) :_j \beta_e$.

Second, we show that if $\nu_e \in \{-, 0\}$ then for all methods $\varsigma(x)b$ such that $\langle j, \varsigma(x)b \rangle \in \beta \rightsquigarrow \beta_e$ we have $\langle j, [m_e{=}\varsigma(x)b, m_c{=}\varsigma(x_c)b_c]_{c \in C \setminus \{e\}} \rangle \in \beta$. Assume that $\nu_e \in \{-, 0\}$, then by the hypothesis (H) we have that $\beta_e \subseteq \alpha_e$, which implies $\lfloor \beta_e \rfloor_k \subseteq \lfloor \alpha_e \rfloor_k$. Let $\langle j, \varsigma(x)b \rangle \in \beta \rightsquigarrow \beta_e$, then since by the induction hypothesis $\lfloor \alpha \rfloor_k \subseteq \lfloor \beta \rfloor_k$ we can apply Lemma 3.4.12 and infer that $\langle j, \varsigma(x)b \rangle \in \alpha \rightsquigarrow \alpha_e$. Since $e \in D$ we can apply (H2) which gives us that $\langle j, [m_e{=}\varsigma(x)b, m_c{=}\varsigma(x_c)b_c]_{c \in C \setminus \{e\}} \rangle \in \alpha$, which by the induction hypothesis allows us to conclude that $\langle j, [m_e{=}\varsigma(x)b, m_c{=}\varsigma(x_c)b_c]_{c \in C \setminus \{e\}} \rangle \in \beta$. □

Lemma 3.4.15 (VarSubObj2). *If for all $d \in D$ we have $\nu_d = 0$ or $\nu_d = \nu'_d$ then $[m_d :_{\nu_d} \tau_d]_{d \in D} \subseteq [m_d :_{\nu'_d} \tau_d]_{d \in D}$.*

Proof. Let us denote $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d \in D}$ and $\alpha' \equiv [m_d :_{\nu'_d} \tau_d]_{d \in D}$. We prove that for all $k \geq 0$ and for all values v if $\langle k, v \rangle \in \alpha$ then $\langle k, v \rangle \in \alpha'$, by course-of-values induction on k. The induction hypothesis is thus that for all $j < k$ if $\langle j, v \rangle \in \alpha$ then $\langle j, v \rangle \in \alpha'$, or equivalently $\lfloor \alpha \rfloor_k \subseteq \lfloor \alpha' \rfloor_k$.

Assuming that $\langle k, v\rangle \in \alpha$ we have to prove that $\langle k, v\rangle \in \alpha'$. By Def. 3.4.1 we have that $v = [m_e{=}\varsigma(x_e)b_e]_{e\in E}$, $D \subseteq E$ and for all $d \in D$ and $j < k$ we have

$$\nu_d \in \{+, 0\} \Rightarrow [x_d \mapsto v]\,(b_d) :_j \tau_d \tag{H1}$$

$$\begin{aligned}\nu_d \in \{-, 0\} \Rightarrow \forall\varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d \Rightarrow& \\ \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha&\end{aligned} \tag{H2}$$

Let $d \in D$ and $j < k$. We first show that if $\nu'_d \in \{+, 0\}$ then $[x_d \mapsto v]\,(b_d) :_j \tau_d$. From the hypothesis we have that $\nu'_d \in \{+, 0\}$ implies $\nu_d \in \{+, 0\}$, which from (H1) implies $[x_d \mapsto v]\,(b_d) :_j \tau_d$.

Second, we show that if $\nu'_d \in \{-, 0\}$ then for all methods $\varsigma(x)b$ such that $\langle j, \varsigma(x)b\rangle \in \alpha' \rightsquigarrow \tau_d$ we have $\langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha'$. Assume that $\nu'_d \in \{-, 0\}$, then using the hypothesis we can infer that $\nu_d \in \{-, 0\}$. Let $\langle j, \varsigma(x)b\rangle \in \alpha' \rightsquigarrow \tau_d$, but by the induction hypothesis $\lfloor\alpha\rfloor_k \subseteq \lfloor\alpha'\rfloor_k$ so we can apply Lemma 3.4.12 which gives us that $\langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d$. From this by (H2) we get that $\langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha$, and by the induction hypothesis also $\langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha'$. □

3.5 Recursive Types

The introduction already presented a recursive type: recursive records. Here we deal with a more general instance of the same idea. Recursive types were the main motivation behind the model of Appel and McAllester [AM01], and their results directly apply here, therefore we do not go into much detail.

Example 3.5.1 (Natural Numbers)**.** Let us return to the encoding of natural numbers given in Section 2.3.3. Numbers were defined as objects with an *iszero* boolean field and two methods yielding the preceding and the succeeding number. Let us examine this more closely: numbers are objects having two methods returning other numbers. This means that the type of numbers contains two methods of number type. The type of numbers is thus a recursive type, which we can write as:

$$\mathit{Nat} \triangleq \mu(\lambda\alpha.\,[\mathit{iszero} : \mathit{Bool}, \mathit{pred} : \alpha, \mathit{succ} : \alpha])$$

The type of *pred* and *succ* is the same as the type of the whole object.

The recursion type operator μ computes a candidate fixpoint of a function from types to types by repeatedly applying the function to $\bot$.

Definition 3.5.2 (Recursion Operator)**.**

$$\mu F = \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\}$$

A type constructor F is contractive if in order to determine whether a term has type $F(\tau)$ with approximation $k + 1$, it suffices to know the type τ only to approximation k.

Definition 3.5.3 (Contractiveness)**.** A type constructor F is *contractive* if for all types τ and for all $k \geq 0$ we have:

$$\lfloor F(\tau)\rfloor_{k+1} = \lfloor F(\lfloor\tau\rfloor_k)\rfloor_{k+1}$$

When additionally F is contractive μ computes a fixpoint of F, that is $\mu F = F(\mu F)$. Recursive types satisfying this condition with equality are called equi-recursive types, in contrast to the iso-recursive types for which μF is only isomorphic to $F(\mu F)$ via two constructs on terms named *fold* and *unfold* [Pie02, Chapters 20 and 21].

Lemma 3.5.4 (Fixpoint)**.** *If F is contractive then $\mu F = F(\mu F)$.*

Proof. Given in [AM01, Theorem 20.] □

The condition $\mu F = F(\mu F)$ immediately gives raise to two semantic typing rules which can only be applied when F is contractive:

$$(\textsc{Unfold})\ \frac{\Sigma \models a : \mu F}{\Sigma \models a : F(\mu F)} \qquad (\textsc{Fold})\ \frac{\Sigma \models a : F(\mu F)}{\Sigma \models a : \mu F}$$

Example 3.5.5 (Natural Numbers: Derivation)**.** Returning to the natural numbers we can now show that $\underline{0}$ has type *Nat.* In Section 2.3.3 we defined $\underline{0}$ as:

$$\underline{0} \triangleq [\mathit{iszero}{=}\mathrm{true}, \mathit{pred}{=}\varsigma(x)x, \mathit{succ}{=}\varsigma(x)[\mathit{iszero}{=}\mathrm{false}, \mathit{pred}{=}x, \mathit{succ}{=}x.\mathit{succ}]]$$

We use the following notations in the derivation below:

$$\tau \equiv [\mathit{iszero} : \mathit{Bool}, \mathit{pred} : \mathit{Nat}, \mathit{succ} : \mathit{Nat}]$$
$$\Sigma \equiv [x \mapsto \tau] \qquad \Sigma' \equiv \Sigma[\_ \mapsto \tau]$$

$$(\textsc{Fold})\ \dfrac{(\textsc{Obj})\ \dfrac{(\textsc{True})\ \Sigma \models \mathrm{true} : \mathit{Bool} \qquad (\textsc{Var})\ \Sigma \models x : \mathit{Nat} \qquad t}{\emptyset \models \underline{0} : [\mathit{iszero} : \mathit{Bool}, \mathit{pred} : \mathit{Nat}, \mathit{succ} : \mathit{Nat}]}}{\emptyset \models \underline{0} : \mu(\lambda\alpha.\, [\mathit{iszero} : \mathit{Bool}, \mathit{pred} : \alpha, \mathit{succ} : \alpha])}$$

$$t \;=\; (\textsc{Fold})\ \dfrac{(\textsc{Obj})\ \dfrac{\Sigma' \models \mathrm{false} : \mathit{Bool} \qquad \Sigma' \models x : \mathit{Nat} \qquad \dfrac{\Sigma' \models x : \tau\ (\textsc{Var})}{\Sigma' \models x.\mathit{succ} : \mathit{Nat}}\ (\textsc{Inv})}{\Sigma \models [\mathit{iszero}{=}\mathrm{false}, \mathit{pred}{=}x, \mathit{succ}{=}x.\mathit{succ}] : \tau}}{\Sigma \models [\mathit{iszero}{=}\mathrm{false}, \mathit{pred}{=}x, \mathit{succ}{=}x.\mathit{succ}] : \mathit{Nat}}$$

Compared to the setting of Appel and McAllester [AM01] we also have a subtyping rule for recursive types, which is known as the Amber rule [Car85]. In order to show that μF is a subtype of μG, it suffices to show that $F(\alpha) \subseteq G(\beta)$ holds for all $\alpha \subseteq \beta$.

$$(\textsc{SubRec})\ \frac{\forall \alpha, \beta \in \mathsf{Type}.\ \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta)}{\mu F \subseteq \mu G}$$

Lemma 3.5.6 (SubRec: Amber Rule)**.** *Given $F, G : \mathsf{Type} \to \mathsf{Type}$, if for all α and β such that $\alpha \subseteq \beta$ we have that $F(\alpha) \subseteq G(\beta)$, then $\mu F \subseteq \mu G$.*

Proof. Under the assumptions of the lemma, we have to show that $\mu F \subseteq \mu G$. By the definition of the recursion operator (Def. 3.5.2) we can equivalently show that $F^{k+1}(\bot) \subseteq G^{k+1}(\bot)$. We do this by induction on k.

Case $k = 0$, since $\bot \subseteq \bot$ by the hypothesis $F(\bot) \subseteq G(\bot)$.

Case $k > 0$, then by the induction hypothesis we have that $F^k(\bot) \subseteq G^k(\bot)$, so by the hypothesis we infer that $F^{k+1}(\bot) \subseteq G^{k+1}(\bot)$.

□

Not all type constructors are contractive, still all the ones we consider in this thesis satisfy a weaker condition called non-expansiveness. A type constructor F is non-expansive if in order to determine whether a term has type $F(\tau)$ to approximation k, it suffices to know the type τ only to approximation k.

Definition 3.5.7 (Non-expansiveness)**.** A type constructor $F : \mathsf{Type} \to \mathsf{Type}$ is *non-expansive* if for all types τ and for all $k \geq 0$ we have:

$$\lfloor F(\tau) \rfloor_k = \lfloor F(\lfloor \tau \rfloor_k) \rfloor_k$$

Lemma 3.5.8. *The object type constructor is contractive.*

$$\left\lfloor [m_d :_{\nu_d} \tau_d]_{d \in D} \right\rfloor_{k+1} = \left\lfloor [m_d :_{\nu_d} \lfloor \tau_d \rfloor_k]_{d \in D} \right\rfloor_{k+1}$$

Proof. It can be shown from the definition of object types (Def. 3.4.1). □

Lemma 3.5.9. *The procedure type constructor is contractive.*

$$\lfloor \alpha \to \beta \rfloor_{k+1} = \lfloor \lfloor \alpha \rfloor_k \to \lfloor \beta \rfloor_k \rfloor_{k+1}$$

Proof. This is immediate by Lemma 3.5.8 since procedure types are encoded using object types as described in Section 3.4.2. □

Counterexample 3.5.10. The recursion operator μ is not contractive. More precisely, there exists F contractive such that $\lfloor \mu F \rfloor_{k+1} \neq \lfloor \mu \lfloor F \rfloor_k \rfloor_{k+1}$. Let $k = 0$ and $F = \lambda\alpha.\, [m : []]$. We have that $\lfloor \mu F \rfloor_{k+1} = \lfloor \mu(\lambda\alpha.\, [m : []]) \rfloor_1 = \lfloor (\lambda\alpha.\, [m : []])(\bot) \rfloor_1 = \lfloor [m : []] \rfloor_1 \neq \bot = \lfloor \bot \rfloor_1 = \lfloor (\lambda\alpha.\, \bot)(\bot) \rfloor_1 = \lfloor \mu(\lambda\alpha.\, \bot) \rfloor_1 = \lfloor \mu(\lambda\alpha.\, \lfloor [m : []] \rfloor_0) \rfloor_1 = \lfloor \mu \lfloor \lambda\alpha.\, [m : []] \rfloor_0 \rfloor_1 = \lfloor \mu \lfloor F \rfloor_k \rfloor_{k+1}$

Lemma 3.5.11. *The recursion operator μ is non-expansive, more precisely if F is contractive then:*

$$\lfloor \mu F \rfloor_k = \lfloor \mu \lfloor F \rfloor_k \rfloor_k$$

Proof. We use the fact that $\lfloor \mu F \rfloor_k = \left\lfloor F^k(\bot) \right\rfloor_k$ [AM01, Lemma 18.(a)] to derive:

$$\lfloor \mu F \rfloor_k = \left\lfloor F^k(\bot) \right\rfloor_k = \left\lfloor \lfloor F \rfloor_k^k (\bot) \right\rfloor_k = \lfloor \mu \lfloor F \rfloor_k \rfloor_k$$

□

3.6 Bounded Quantified Types

3.6.1 Vestigial Operators

Example 3.6.1 (Booleans)**.** In Section 2.3.2 we gave the Church encoding for booleans. True was encoded as $\lambda x.\, \lambda y.\, x$ and false as $\lambda x.\, \lambda y.\, y$. Both true and false are procedures taking two arguments, and returning one of them. For every type α true and false can be given the type $\alpha \to \alpha \to \alpha$, however we want to capture this into just one type for booleans. We can do this using a universal type (also known as polymorphic type) [Pie02, Chapter 23]:

$$\mathit{Bool} \triangleq \forall(\lambda \alpha.\, \alpha \to \alpha \to \alpha)$$

We would expect that in an untyped setting $\lambda x.\, \lambda y.\, x$ and $\lambda x.\, \lambda y.\, y$ are the values of type *Bool*. However, this is unfortunately not the case. Our model requires us to have introduction and elimination forms for quantified types, just like in a typed setting. In a typed setting type abstraction would be written as $\Lambda X.a$ where X is a type variable, and type application as $(a\ A)$ where A is a type expression. Applying a type abstraction to a type expression causes the expression to be substituted for the type variable in the body of the abstraction $(\Lambda X.a)\ A \to a[X{:=}A]$. Our type abstraction and application operators can be viewed as reminiscents of the real operators after all types were removed, therefore we call them vestigial operators. Polymorphic values are introduced by a type abstraction construct "$\Lambda.\, a$" and eliminated by a type application construct "$a\,[]$". The boolean terms are redefined as follows:

$$\text{true} \triangleq \Lambda.\, \lambda x.\, \lambda y.\, x \qquad \text{false} \triangleq \Lambda.\, \lambda x.\, \lambda y.\, y$$

$$\text{if } b \text{ then } a_1 \text{ else } a_2 \triangleq b\,[]\ a_1\ a_2$$

Our syntax of terms has to accommodate four new forms, the two for polymorphism (universal types) discussed above, and pack and open for type abstraction (existential types).

$$a, b ::= \ldots \mid \Lambda.\, a \mid a\,[] \mid \text{pack } a \mid \text{open } a \text{ as } x \text{ in } b$$

There are also new kinds of values, new reduction rules and contexts:

$$v \in \mathsf{Val} ::= \ldots \mid \Lambda.\, a \mid \text{pack } v$$

$$(\textsc{Red-TApp})\ (\Lambda.\, a)\,[] \to a$$

$$(\textsc{Red-Open})\ \text{open (pack } v) \text{ as } x \text{ in } b \to [x \mapsto v]\,(b)$$

$$C[\bullet] ::= \bullet \mid \ldots \mid C\,[] \mid \text{pack } C \mid \text{open } C \text{ as } x \text{ in } b$$

3.6.2 Definition and Rules

We consider both universal and existential impredicative quantified types. This is the most powerful kind of quantified types, since the quantification is over all types, including the quantified types themselves. In particular this is more expressive then the *let*-polymorphism in ML where the quantifiers only range over monotypes, which do not contain quantifiers. Impredicative quantified

$$(\textsc{TAbs})\ \frac{\forall\tau{\in}\mathsf{Type}.\ \tau\subseteq\alpha\Rightarrow\Sigma\models a:F(\tau)}{\Sigma\models\Lambda.\,a:\forall_\alpha F}$$

$$(\textsc{TApp})\ \frac{\Sigma\models a:\forall_\alpha F\quad \tau{\in}\mathsf{Type}\quad \tau\subseteq\alpha}{\Sigma\models a\,[]:F(\tau)}$$

$$(\textsc{Pack})\ \frac{\exists\tau{\in}\mathsf{Type}.\ \tau\subseteq\alpha\ \wedge\ \Sigma\models a:F(\tau)}{\Sigma\models \text{pack}\ a:\exists_\alpha F}$$

$$(\textsc{Open})\ \frac{\Sigma\models a:\exists_\alpha F\quad \forall\tau{\in}\mathsf{Type}.\ \tau\subseteq\alpha\Rightarrow\Sigma[x\mapsto F(\tau)]\models b:\beta}{\Sigma\models \text{open}\ a\ \text{as}\ x\ \text{in}\ b:\beta}$$

$$(\textsc{SubUniv})\ \frac{\beta\subseteq\alpha\quad \forall\tau\in\mathsf{Type}.\ \tau\subseteq\beta\Rightarrow F(\tau)\subseteq G(\tau)}{\forall_\alpha F\subseteq\forall_\beta G}$$

$$(\textsc{SubExist})\ \frac{\alpha\subseteq\beta\quad \forall\tau\in\mathsf{Type}.\ \tau\subseteq\alpha\Rightarrow F(\tau)\subseteq G(\tau)}{\exists_\alpha F\subseteq\exists_\beta G}$$

Figure 3.3: Semantic typing rules for bounded quantified types

types were previously studied by Ahmed [Ahm04] for a λ-calculus with general references, so we follow her presentation and simplify whenever possible. However, unlike in the work of Ahmed our quantifiers have bounds (bounded quantification) and we are also studying subtyping.

Definition 3.6.2 (Bounded Quantified Types)**.** Let F non-expansive

$$\forall_\alpha F \triangleq \{\langle k,\Lambda.\,a\rangle \mid \forall\tau.\ \lfloor\tau\rfloor_k{\in}\mathsf{Type}\ \wedge\ \lfloor\tau\rfloor_k\subseteq\lfloor\alpha\rfloor_k\Rightarrow\forall j{<}k.\ a:_j F(\tau)\}$$

$$\exists_\alpha F \triangleq \{\langle k,\text{pack}\ v\rangle \mid \exists\tau.\ \lfloor\tau\rfloor_k{\in}\mathsf{Type}\ \wedge\ \lfloor\tau\rfloor_k\subseteq\lfloor\alpha\rfloor_k\ \wedge\ \forall j{<}k.\ \langle j,k\rangle\in F(\tau)\}$$

Lemma 3.6.3. *If α is a type and F is a non-expansive function from types to types, then $\forall_\alpha F$ and $\exists_\alpha F$ are also types.*

Proof. Immediate from Def. 3.6.2. □

The typing rules in Figure 3.3 are resembling the second-order type system for the functional object calculus of Abadi and Cardelli [AC95b].

Lemma 3.6.4 (TAbs)**.** *Let F be a non-expansive function from types to types. If for all types τ such that $\tau\subseteq\alpha$ we have $\Sigma\models a:F(\tau)$ then $\Sigma\models\Lambda.\,a:\forall_\alpha F$.*

Proof. Let $k\geq 0$ and $\sigma:_k\Sigma$. We want to show that $\Lambda.\,\sigma(a):_k\forall_\alpha F$, so let τ be a set such that $\lfloor\tau\rfloor_k\in\mathsf{Type}$ and $\lfloor\tau\rfloor_k\subseteq\lfloor\alpha\rfloor_k$. Since $\lfloor\alpha\rfloor_k\subseteq\alpha$ by transitivity $\lfloor\tau\rfloor_k\subseteq\alpha$. From the hypothesis we thus get that $a(\sigma):_k F(\lfloor\tau\rfloor_k)$, so by the non-expansiveness of F it follows that for all $j<k$ we have $\sigma(a):_j F(\tau)$. By Def. 3.6.2 we conclude that $\Lambda.\,\sigma(a):_k\forall_\alpha F$. □

Lemma 3.6.5 (TAbs). *Let F be a non-expansive function from types to types, α and τ two types such that $\tau \subseteq \alpha$. If $\Sigma \models a : \forall_\alpha F$ then $\Sigma \models a\,[\,] : F(\tau)$.*

Proof. Let $k \geq 0$ and $\sigma :_k \Sigma$. By Def. 3.1.8 it suffices to show that $\sigma(a)\,[\,] :_k F(\tau)$. Let $j < k$ such that $\sigma(a)\,[\,] \to b$ and $b \not\to$. Since from the hypothesis by Def. 3.1.8 it follows that $\sigma(a) :_k \forall_\alpha F$, we can apply Lemma 3.1.10 and get that $\sigma(a)$ is safe for k steps. Thus by the operational semantics we infer that $\exists i < j$ such that $\sigma(a) \to^i v$ and v is a value thus irreducible. Since $\sigma(a) :_k \forall_\alpha F$ by Def. 3.1.4 we get that $\langle k - i, v\rangle \in \forall_\alpha F$. From the hypothesis τ is a type and $\tau \subseteq \alpha$, so $\lfloor\tau\rfloor_k$ is also a type and also $\lfloor\tau\rfloor_k \subseteq \lfloor\alpha\rfloor_k$. We can thus apply the definition of the boundend universal type (Def. 3.6.2) to get that $v = \Lambda.\, b'$ and $b' :_{k-i-1} F(\lfloor\tau\rfloor_k)$, which is equivalent to $b' :_{k-i-1} F(\tau)$ since F is non-expansive. By the rules Red-Ctx and Red-TApp we also get that $\sigma(a)\,[\,] \to^i \Lambda.\, b'\,[\,] \to b' \to^{j-i-1} b$, so by applying Def. 3.1.4 we first get that $\langle k - j, b\rangle \in F(\tau)$ and from this we conclude that $\sigma(a)\,[\,] :_k F(\tau)$. □

Lemma 3.6.6 (Pack). *Let F be a non-expansive function from types to types and α a type. If there exists a type τ such that $\tau \subseteq \alpha$ and $\Sigma \models a : F(\tau)$ then $\Sigma \models pack\, a : \exists_\alpha F$.*

Proof. Similar to the proof of Lemma 3.6.4 (type abstraction). □

Lemma 3.6.7 (Open). *Let F be a non-expansive function from types to types. If $\Sigma \models a : \exists_\alpha F$ and for all types tau such that $\tau \subseteq \alpha$ we have $\Sigma[x \mapsto F(\tau)] \models b : \beta$ then $\Sigma \models open\, a\ as\ x\ in\, b : \beta$.*

Proof. Similar to the proof of Lemma 3.6.5 (type application). □

Lemma 3.6.8 (SubUniv). *Let F and G be two non-expansive functions from types to types, and α and β two types such that $\beta \subseteq \alpha$. If for all types $\tau \subseteq \beta$ we have that $F(\tau) \subseteq G(\tau)$ then $\forall_\alpha F \subseteq \forall_\beta G$.*

Proof. We assume that $\langle k, \Lambda.\, a\rangle \in \forall_\alpha F$ and show that $\langle k, \Lambda.\, a\rangle \in \forall_\beta G$. Let τ be a set such that $\lfloor\tau\rfloor_k \in \mathsf{Type}$ and $\lfloor\tau\rfloor_k \subseteq \lfloor\beta\rfloor_k$. From the hypothesis $\beta \subseteq \alpha$ so also $\lfloor\beta\rfloor_k \subseteq \lfloor\alpha\rfloor_k$, and thus by transitivity $\lfloor\tau\rfloor_k \subseteq \lfloor\alpha\rfloor_k$. Let $j < k$, since $\langle k, \Lambda.\, a\rangle \in \forall_\alpha F$ by the definition of the universal type (Def. 3.6.2) we get that $a :_j F(\tau)$. Since F is non-expansive this is equivalent to $a :_j F(\lfloor\tau\rfloor_k)$. But $\lfloor\tau\rfloor_k \in \mathsf{Type}$ and $\lfloor\tau\rfloor_k \subseteq \lfloor\beta\rfloor_k \subseteq \beta$, so by the hypothesis we infer that $F(\lfloor\tau\rfloor_k) \subseteq G(\lfloor\tau\rfloor_k)$. This means that $a :_j G(\lfloor\tau\rfloor_k)$ and equivalently that $a :_j G(\tau)$. By Def. 3.6.2 it follows that $\langle k, \Lambda.\, a\rangle \in \forall_\beta G$. □

Lemma 3.6.9 (SubExist). *Let F and G be two non-expansive functions from types to types, and α and β two types such that $\alpha \subseteq \beta$. If for all types $\tau \subseteq \alpha$ we have that $F(\tau) \subseteq G(\tau)$ then $\exists_\alpha F \subseteq \exists_\beta G$.*

Proof. We assume that $\langle k, \text{pack } v\rangle \in \exists_\alpha F$ and show that $\langle k, \text{pack } v\rangle \in \exists_\beta G$. From this assumption by Def. 3.6.2 there exists a set τ such that $\lfloor\tau\rfloor_k \in \mathsf{Type}$ and $\lfloor\tau\rfloor_k \subseteq \lfloor\alpha\rfloor_k$ and for all $j < k$ we have $\langle j, v\rangle \in F(\tau)$, or equivalently since F is contractive $\langle j, v\rangle \in F(\lfloor\tau\rfloor_k)$. Since by the hypothesis $\alpha \subseteq \beta$ we also have that $\lfloor\alpha\rfloor_k \subseteq \lfloor\beta\rfloor_k$, so by transitivity also $\lfloor\tau\rfloor_k \subseteq \lfloor\beta\rfloor_k$.

Let $j < k$, as shown above $\langle j, v\rangle \in F(\lfloor\tau\rfloor_k)$. Additionally we have that $\lfloor\tau\rfloor_k \subseteq \lfloor\alpha\rfloor_k \subseteq \alpha$ so by the hypothesis $F(\lfloor\tau\rfloor_k) \subseteq G(\lfloor\tau\rfloor_k)$. Thus $\langle j, v\rangle \in G(\lfloor\tau\rfloor_k)$, and since G is non-expansive also $\langle j, v\rangle \in G(\tau)$. This together with $\lfloor\tau\rfloor_k \in \mathsf{Type}$ and $\lfloor\tau\rfloor_k \subseteq \lfloor\beta\rfloor_k$ allow us to conclude that $\langle k, \text{pack } v\rangle \in \exists_\beta G$. □

From the definitions we can also easily show that the universal and existential type constructors are contractive.

Lemma 3.6.10. *The boundend universal and existential type constructors are contractive.*

$$\lfloor \forall_\alpha F \rfloor_{k+1} = \left\lfloor \forall_{\lfloor \alpha \rfloor_k} \lfloor F \rfloor_k \right\rfloor_{k+1}$$
$$\lfloor \exists_\alpha F \rfloor_{k+1} = \left\lfloor \exists_{\lfloor \alpha \rfloor_k} \lfloor F \rfloor_k \right\rfloor_{k+1}$$

3.7 Semantic Typing Rules

In this section we repeat the semantic typing rules that we will use in the remainder of this thesis. They define an interface that hides the internals of the model, and in fact our model for the imperative object calculus also satisfies this interface [HS07].

Subtyping $\boxed{\alpha \subseteq \beta}$

$$(\textsc{SubRefl})\ \alpha \subseteq \alpha \qquad (\textsc{SubTrans})\ \frac{\alpha \subseteq \tau \quad \tau \subseteq \beta}{\alpha \subseteq \beta}$$

$$(\textsc{SubTop})\ \alpha \subseteq \top \qquad (\textsc{SubBot})\ \bot \subseteq \alpha$$

$$(\textsc{SubProc})\ \frac{\alpha' \subseteq \alpha' \quad \beta \subseteq \beta'}{\alpha \to \beta \subseteq \alpha' \to \beta'}$$

$$(\textsc{VarSubObj1})\ \frac{\begin{array}{r} E \subseteq D \quad \forall e \in E.\, (\nu_e \in \{+,0\} \Rightarrow \alpha_e \subseteq \beta_e) \\ \wedge\, (\nu_e \in \{-,0\} \Rightarrow \beta_e \subseteq \alpha_e) \end{array}}{[m_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [m_e :_{\nu_e} \beta_e]_{e \in E}}$$

$$(\textsc{VarSubObj2})\ \frac{\forall d \in D.\, \nu_d = 0 \ \vee\ \nu_d = \nu'_d}{[m_d :_{\nu_d} \tau_d]_{d \in D} \subseteq [m_d :_{\nu'_d} \tau_d]_{d \in D}}$$

$$(\textsc{SubRec})\ \frac{\forall \alpha, \beta \in \mathsf{Type}.\ \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta)}{\mu F \subseteq \mu G}$$

$$(\textsc{SubUniv})\ \frac{\beta \subseteq \alpha \quad \forall \tau \in \mathsf{Type}.\ \tau \subseteq \beta \Rightarrow F(\tau) \subseteq G(\tau)}{\forall_\alpha F \subseteq \forall_\beta G}$$

$$(\textsc{SubExist})\ \frac{\alpha \subseteq \beta \quad \forall \tau \in \mathsf{Type}.\ \tau \subseteq \alpha \Rightarrow F(\tau) \subseteq G(\tau)}{\exists_\alpha F \subseteq \exists_\beta G}$$

Figure 3.4: Semantic typing rules for subtyping

$$\boxed{\Sigma \models a : \alpha}$$

$$(\textsc{Var})\ \Sigma \models x : \Sigma(x) \qquad (\textsc{Sub})\ \frac{\Sigma \models a : \alpha \quad \alpha \subseteq \beta}{\Sigma \models a : \beta}$$

Procedure Types

$$(\textsc{Lam})\ \frac{\Sigma[x \mapsto \alpha] \models b : \beta}{\Sigma \models \lambda x.\, b : \alpha \to \beta} \qquad (\textsc{App})\ \frac{\Sigma \models a : \beta \to \alpha \quad \Sigma \models b : \beta}{\Sigma \models a\, b : \alpha}$$

Object Types Let $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d \in D}$

$$(\textsc{VarObj})\ \frac{\forall d \in D.\ \Sigma[x \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha}$$

$$(\textsc{VarInv})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \nu_e \in \{+, 0\}}{\Sigma \models a.m_e : \tau_e}$$

$$(\textsc{VarUpd})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \nu_e \in \{-, 0\} \quad \Sigma[x \mapsto \alpha] \models b : \tau_e}{\Sigma \models a.m_e \coloneqq \varsigma(x)b : \alpha}$$

Recursive Types If F contractive then

$$(\textsc{Unfold})\ \frac{\Sigma \models a : \mu F}{\Sigma \models a : F(\mu F)} \qquad (\textsc{Fold})\ \frac{\Sigma \models a : F(\mu F)}{\Sigma \models a : \mu F}$$

Bounded Quantified Types

$$(\textsc{TAbs})\ \frac{\forall \tau{\in}\mathsf{Type}.\ \tau \subseteq \alpha \Rightarrow \Sigma \models a : F(\tau)}{\Sigma \models \Lambda.\, a : \forall_\alpha F}$$

$$(\textsc{TApp})\ \frac{\Sigma \models a : \forall_\alpha F \quad \tau{\in}\mathsf{Type} \quad \tau \subseteq \alpha}{\Sigma \models a\,[\,] : F(\tau)}$$

$$(\textsc{Pack})\ \frac{\exists \tau{\in}\mathsf{Type}.\ \tau \subseteq \alpha \ \wedge\ \Sigma \models a : F(\tau)}{\Sigma \models \text{pack}\ a : \exists_\alpha F}$$

$$(\textsc{Open})\ \frac{\Sigma \models a : \exists_\alpha F \quad \forall \tau{\in}\mathsf{Type}.\ \tau \subseteq \alpha \Rightarrow \Sigma[x \mapsto F(\tau)] \models b : \beta}{\Sigma \models \text{open}\ a\ \text{as}\ x\ \text{in}\ b : \beta}$$

Figure 3.5: Semantic typing rules for object, procedure, recursive and bounded quantified types

Chapter 4

Syntactic Types

The semantic type system from Chapter 3 is sound, but type checking ς-terms with respect to it is clearly undecidable. One important reason for the undecidability is that type variables are not explicitly considered in the semantic typing rules for recursive and quantified types. The preconditions of these rules are therefore expressed by quantifying over the infinite set of all types.

One possible solution was given by Swadi, who added type variables to a step-indexed semantic model and used a semantic kind system to track the contractiveness and non-expansiveness of types with free type variables [ARS02, Swa03].

We take a different approach that allows us to keep the semantic model simple, and still get decidable type checking. We introduce a syntactic type system for the functional object calculus that tracks type variables and syntactically enforces the contractiveness of the type expressions used with recursive types. We prove the soundness of the syntactic type system with respect to the semantic model presented in Chapter 3.

The syntactic type system works on terms with type annotations, and it has the minimal type property, but it does not have intersection types since they are not required by our high-level calculus. It has iso-recursive types which are straightforward to implement compared to the equi-recursive types we considered in the semantic model. Finally, the syntactic type system also has bounded quantification. Pierce has shown that even in a syntactic setting with fully annotated terms bounded quantification is undecidable, but also that it can be easily made decidable by adding a simple syntactic restriction on the subtyping rule for bounded universal types [Pie94].

4.1 Revised Syntax of Terms

The operational and step-indexed semantics of the ς-calculus were defined with respect to an untyped syntax of terms. However, type annotations are very useful for guiding type checking, rendering it tractable in practice. For this reason we only type-check fully annotated programs according to the syntax given in Figure 4.1. Note that in order to avoid considering encodings again we added first-class procedures to the language.

As in many compilers for practical programming languages, type annotations

$$\begin{array}{rcll}
a, b & ::= & x & \text{(variable)} \\
& | & [m_d{=}\varsigma(x_d{:}A)b_d]_{d \in D} & \text{(object creation)} \\
& | & a.m & \text{(method invocation)} \\
& | & a.m := \varsigma(x{:}A)b & \text{(method update)} \\
& | & \lambda x{:}A.\, b & \text{(procedure)} \\
& | & a\ b & \text{(procedure application)} \\
& | & \mathrm{unfold}_{\mu X.A}\ a & \text{(recursive type unfold)} \\
& | & \mathrm{fold}_{\mu X.A}\ a & \text{(recursive type fold)} \\
& | & \Lambda X{\leqslant}A.\, b & \text{(type abstraction)} \\
& | & a\ A & \text{(type application)} \\
& | & \mathrm{pack}\ X{\leqslant}A = C\ \mathrm{in}\ a{:}B & \text{(creating existential package)} \\
& | & \mathrm{open}\ a\ \mathrm{as}\ X{\leqslant}A, x{:}B\ \mathrm{in}\ b{:}D & \text{(opening package)}
\end{array}$$

Figure 4.1: Syntax of terms with type annotations

are discarded after type-checking, and do not have any computational effect [Pie02]. Type erased programs are evaluated under the operational semantics given in Chapter 2. The main technical result of this chapter is that well-typed programs are guaranteed to evaluate safely once type-erased.

4.2 Syntactic Type System

In order to have a sound type system, we need to reject meaningless recursive types like $\mu X.X$, and to enforce that the meaning of recursive types is really that of a fixpoint operator on types. Our semantic rules for recursive types state that μF is a fixpoint of F, only under the condition that F is contractive (Figure 3.5). As we have previously shown the type constructors we use here are contractive (Lemmas 3.5.8, 3.5.9 and 3.6.10), with the exception of the identity and the recursion operator itself (Counterexample 3.5.10), which are however still non-expansive (Lemma 3.5.11). Therefore, we enforce contractiveness by grouping type constructors into two syntactical categories.

Definition 4.2.1 (Syntactic Types)**.**

$A, B \in \mathsf{SynTypes} ::= \underline{A} \mid X \mid \mu X.\underline{A}$

$\underline{A}, \underline{B} ::= \mathit{Top} \mid \mathit{Bot} \mid A{\to}B \mid [m_d :_{\nu_d} A_d]_{d \in D} \mid \forall X{\leqslant}A.B \mid \exists X{\leqslant}A.B$

where $\nu ::= 0 \mid + \mid -$

The types denoted by underlined letters ($\underline{A}, \underline{B}$) are guaranteed to be contractive, and therefore can be used in expressions of the form $\mu X.\underline{A}$.

Definition 4.2.2 (Type Environments)**.**

$$\Gamma = \cdot \mid \Gamma, x{:}A \mid \Gamma, X{\leqslant}A$$

The features of our semantic type system closely mirror all the ones we had in the semantic setting. We consider object types with variance annotations, procedure types, recursive types, bounded quantified types and subtyping.

The type system consists of four typing judgements:

$\Gamma \vdash \diamond$	Well-formed Environment (Figure 4.2)
$\Gamma \vdash A$	Well-formed Type (Figure 4.2)
$\Gamma \vdash A \leqslant B$	Subtyping (Figure 4.3)
$\Gamma \vdash a : A$	Term Typing (Figure 4.4)

The inference rules defining our judgements are entirely standard [AC96a, Car97, Pie02]. A type environment is well-formed, if it does not contain any duplicate bindings for variables and type variables (Figure 4.2). A type is well-formed with respect to an environment Γ, if all its type variables are bound by Γ (Figure 4.2).

The subtyping relation is defined by the rules in Figure 4.3, which directly correspond to the semantic typing rules in Figure 3.5. In particular SynSub-Obj1 and SynSubObj2 allow subtyping object types in width and in depth.

One surprising fact about the subtyping rule for bounded universals (Syn-SubUniv) is that it causes type checking to be undecidable in general. As shown by Pierce restricting this rule to universals with equal bounds (as in SynSub-Univ' below) makes type checking decidable, at the cost of rejecting some safe programs [Pie94].

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

Actually, even without this restriction the type checking semi-algorithms used for similar type systems are efficient in practice on all interesting programs, and are guaranteed to terminate on all well-typed ones [CW85, Pie94].

The rules defining the syntactic typing relation for terms are given in Figure 4.4. The type annotations of the terms can be used to effectively guide type checking, and they guarantee the existence of a minimal type for each typable term (although in this setting we do not have intersection types). The unannotated term from Example 3.3.10, needs to be written now either as $[m{=}\varsigma(x{:}[m : [m : []]])[m{=}\varsigma(y{:}[m : []])[]]]$ in which case it has $[m : [m : []]]$ as a minimal type, or as $[m{=}\varsigma(x{:}[m : []])[m{=}\varsigma(y{:}m : [])[]]]$ in which case it has $[m : []]$ as a minimal type.

One other important difference to the semantic type system is that here we are considering iso-recursive rather than equi-recursive types, so folds and unfolds are tagged accordingly in the syntax of terms (Figure 4.1). In the soundness proof the syntactic iso-recursive types can be reduced to the semantic equi-recursive ones, since type erasure also removes folds and unfolds (Appendix A.3). However, iso-recursive types are more straightforward to implement, especially in the presence of subtyping [Pie02, Chapter 21].

Well-formed Environment $\boxed{\Gamma \vdash \diamond}$

$$(\textsc{EnvEmpty})\ \cdot \vdash \diamond \qquad (\textsc{EnvVar})\ \frac{\Gamma \vdash A \quad x \notin \mathit{Vars}(\Gamma)}{\Gamma, x{:}A \vdash \diamond}$$

$$(\textsc{EnvTypeVar})\ \frac{\Gamma \vdash A \quad X \notin \mathit{TypeVars}(\Gamma)}{\Gamma, X{\leqslant}A \vdash \diamond}$$

Well-formed Type $\boxed{\Gamma \vdash A}$

$$(\textsc{TypeTop})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Top}} \qquad (\textsc{TypeBot})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Bot}}$$

$$(\textsc{TypeVar})\ \frac{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash \diamond}{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash X} \qquad (\textsc{TypeProc})\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A{\rightarrow}B}$$

$$(\textsc{TypeObj})\ \frac{\forall d{\in}D.\ \Gamma \vdash A_d}{\Gamma \vdash [m_d{:}A_d]_{d\in D}} \qquad (\textsc{TypeRec})\ \frac{\Gamma, X{\leqslant}\mathit{Top} \vdash \underline{A}}{\Gamma \vdash \mu X.\underline{A}}$$

$$(\textsc{TypeUniv})\ \frac{\Gamma, X{\leqslant}A \vdash B}{\Gamma \vdash \forall X{\leqslant}A.B} \qquad (\textsc{TypeExist})\ \frac{\Gamma, X{\leqslant}A \vdash B}{\Gamma \vdash \exists X{\leqslant}A.B}$$

Figure 4.2: Well-formed syntactic types and type environments

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

$$(\textsc{SynSubRefl})\ \frac{\Gamma \vdash A}{\Gamma \vdash A \leqslant A} \qquad (\textsc{SynSubTrans})\ \frac{\Gamma \vdash A \leqslant A' \quad \Gamma \vdash A' \leqslant B}{\Gamma \vdash A \leqslant B}$$

$$(\textsc{SynSubTop})\ \frac{\Gamma \vdash A}{\Gamma \vdash A \leqslant \mathit{Top}} \qquad (\textsc{SynSubBot})\ \frac{\Gamma \vdash A}{\Gamma \vdash \mathit{Bot} \leqslant A}$$

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

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

$$(\textsc{SynSubObj1})\ \frac{\begin{array}{r} E \subseteq D \quad \forall e {\in} E.\ (\nu_e \in \{+, 0\} \Rightarrow \Gamma \vdash A_e \leqslant B_e) \\ \wedge\ (\nu_e \in \{-, 0\} \Rightarrow \Gamma \vdash B_e \leqslant A_e) \end{array}}{\Gamma \vdash [m_d :_{\nu_d} A_d]_{d \in D} \leqslant [m_e :_{\nu_e} B_e]_{e \in E}}$$

$$(\textsc{SynSubObj2})\ \frac{\forall d \in D.\ \nu_d = 0\ \vee\ \nu_d = \nu'_d}{\Gamma \vdash [m_d :_{\nu_d} A_d]_{d \in D} \leqslant [m_d :_{\nu'_d} A_d]_{d \in D}}$$

$$(\textsc{SynSubRec})\ \frac{\Gamma \vdash \mu X.\underline{A} \quad \Gamma \vdash \mu Y.\underline{B} \quad \Gamma, Y {\leqslant} \mathit{Top}, X {\leqslant} Y \vdash \underline{A} \leqslant \underline{B}}{\Gamma \vdash \mu X.\underline{A} \leqslant \mu Y.\underline{B}}$$

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

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

Figure 4.3: Syntactic subtyping

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

$$(\textsc{SynSub})\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash a : B} \qquad (\textsc{SynVar})\ \frac{\Gamma_1, x{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, x{:}A, \Gamma_2 \vdash x : A}$$

Procedure Types

$$(\textsc{SynLam})\ \frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda x{:}A.\, b : A \to B} \qquad (\textsc{SynApp})\ \frac{\Gamma \vdash a : B \to A \quad \Gamma \vdash b : B}{\Gamma \vdash a\ b : A}$$

Object Types Let $A \equiv [m_d :_{\nu_d} A_d]_{d \in D}$

$$(\textsc{SynObj})\ \frac{\forall d{\in}D.\ \Gamma, x{:}A \vdash b_d : A_d}{\Gamma \vdash [m_d{=}\varsigma(x_d{:}A)b_d]_{d\in D} : A}$$

$$(\textsc{SynInv})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \nu_e \in \{+, 0\}}{\Gamma \vdash a.m_e : A_e}$$

$$(\textsc{SynUpd})\ \frac{\Gamma \vdash a : A \quad e \in D \quad \Gamma, x{:}A \vdash b : A_e \quad \nu_e \in \{-, 0\}}{\Gamma \vdash a.m_e \coloneqq \varsigma(x{:}A)b : A}$$

Recursive Types

$$(\textsc{SynUnfold})\ \frac{\Gamma \vdash a : \mu X.\underline{A}}{\Gamma \vdash \mathrm{unfold}_{\mu X.\underline{A}}\, a : \underline{A}[X{:=}\mu X.\underline{A}]}$$

$$(\textsc{SynFold})\ \frac{\Gamma \vdash a : \underline{A}[X{:=}\mu X.\underline{A}]}{\Gamma \vdash \mathrm{fold}_{\mu X.\underline{A}}\, a : \mu X.\underline{A}}$$

Bounded Quantified Types

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

$$(\textsc{SynTApp})\ \frac{\Gamma \vdash a : \forall X{\leqslant}A.B \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash a\ A' : B[X{:=}A']}$$

$$(\textsc{SynPack})\ \frac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash a[X{:=}C] : B[X{:=}C]}{\Gamma \vdash \text{pack } X{\leqslant}A = C \text{ in } a{:}B : \exists X{\leqslant}A.B}$$

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

Figure 4.4: Syntactic typing of terms

4.3 Semantic Soundness

In order to prove our syntactic type system sound, we relate our syntactic types to their semantic counterparts, for which we have already proved soundness in Theorem 3.1.12. This approach is standard in denotational semantics [Mil78, Win93, Mit96, AC96a]; what is maybe surprising is the expressive power of the step-indexed semantic model.

Definition 4.3.1 (The Meaning of Types)**.** Let η be a finite map from type variables to semantic types. If A is a syntactic type that only references type variables in the domain of η, then its meaning with respect to η is a semantic type given by the following meaning function, which is recursively defined on the structure of A.

$$[\![Top]\!]_\eta = \top$$

$$[\![Bot]\!]_\eta = \bot$$

$$[\![X]\!]_\eta = \eta(X)$$

$$[\![A \to B]\!]_\eta = [\![A]\!]_\eta \to [\![B]\!]_\eta$$

$$\left[\!\left[[m_d :_{\nu_d} A_d]_{d\in D} \right]\!\right]_\eta = \left[m_d :_{\nu_d} [\![A_d]\!]_\eta \right]_{d\in D}$$

$$[\![\mu X.\underline{A}]\!]_\eta = \mu(\lambda\alpha{\in}\mathsf{Type}.\ [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]})$$

$$[\![\forall X{\leqslant}A.B]\!]_\eta = \forall_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathsf{Type}.\ [\![\eta[X\mapsto\alpha]]\!]_{)}$$

$$[\![\exists X{\leqslant}A.B]\!]_\eta = \exists_{[\![A]\!]_\eta}(\lambda\alpha{\in}\mathsf{Type}.\ [\![B]\!]_{\eta[X\mapsto\alpha]})$$

Property 4.3.2. *If $X{\notin}FV(A)$, then $[\![A]\!]_{\eta[X\mapsto\alpha]} = [\![A]\!]_\eta$.*

Definition 4.3.3 ($\eta \models \Gamma$)**.** If Γ is a well-formed type environment that only contains type variables in η, then we say that η satisfies Γ if and only if the following relation defined on the structure of Γ holds.

$$\emptyset \models \cdot \qquad \frac{\eta \models \Gamma}{\eta \models \Gamma, x{:}A} \qquad \frac{\eta \models \Gamma \quad \eta(X) \subseteq [\![A]\!]_\eta}{\eta \models \Gamma, X{\leqslant}A}$$

Property 4.3.4 (Prefix)**.** *If $\eta \models \Gamma_1, \Gamma_2$, then $\eta \models \Gamma_1$*

Definition 4.3.5 (Meaning of Type Environments)**.** Given a map η and a well-formed syntactic type environment Γ that only contains type variables in η, then the meaning of Γ with respect to η is the semantic type environment given by the following function defined on the structure of Γ.

$$[\![\cdot]\!]_\eta = \emptyset$$

$$[\![\Gamma, X{\leqslant}A]\!]_\eta = [\![\Gamma]\!]_\eta$$

$$[\![\Gamma, x{:}A]\!]_\eta = [\![\Gamma]\!]_\eta \left[x \mapsto [\![A]\!]_\eta \right]$$

We start by showing that the two syntactic categories of type expressions correspond in fact to contractive and non-expansive functions in the model. Then we prove a substitution lemma capturing the interaction between syntactic type substitution and the meaning function. Those auxiliary results allow us to prove the soundness of the subtyping relation, and ultimately the semantic soundness of the syntactic type system with respect to the semantic model, which immediately implies its type safety.

Lemma 4.3.6 (Contractiveness and Non-expansiveness)**.**
For all A and $\underline{A}$ such that $FV(\underline{A}) \subseteq Dom(\eta)$ and $FV(A) \subseteq Dom(\eta)$ we have:

1. *$[\![\underline{A}]\!]$ is contractive in η.*
2. *$[\![A]\!]$ is non-expansive in η*

Proof. By mutual induction on the structure of A and $\underline{A}$.

1. We first show that $\left\lfloor [\![\underline{A}]\!]_\eta \right\rfloor_{k+1} = \left\lfloor [\![\underline{A}]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_{k+1}$ by case analysis on the structure of $\underline{A}$.

Case $\underline{A} = Top$ or $\underline{A} = Bot$. Immediate.

Case $\underline{A} = A{\to}B$.

$$\begin{aligned}
\left\lfloor [\![\underline{A}]\!]_\eta \right\rfloor_{k+1} &= \left\lfloor [\![A]\!]_\eta \to [\![B]\!]_\eta \right\rfloor_{k+1} && \text{(by Def. 4.3.1)}\\
&= \left\lfloor \left\lfloor [\![A]\!]_\eta \right\rfloor_k \to \left\lfloor [\![B]\!]_\eta \right\rfloor_k \right\rfloor_{k+1} && \text{(by Lemma 3.5.9)}\\
&= \left\lfloor \left\lfloor [\![A]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_k \to \left\lfloor [\![B]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_k \right\rfloor_{k+1} && \text{(induction hyp.)}\\
&= \left\lfloor [\![A]\!]_{\lfloor\eta\rfloor_k} \to [\![B]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_{k+1} && \text{(by Lemma 3.5.9)}\\
&= \left\lfloor [\![\underline{A}]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_{k+1} && \text{(by Def. 4.3.1)}
\end{aligned}$$

Case $\underline{A} = [m_d :_{\nu_d} A_d]_{d\in D}$. Similarly, from Lemma 3.5.8 and the induction hypothesis.

Case $\underline{A} = \forall X{\leqslant}A.B$.

$$\begin{aligned}
\left\lfloor [\![\underline{A}]\!]_\eta \right\rfloor_{k+1} &= \left\lfloor \forall_{[\![A]\!]_\eta}(\lambda\alpha.\ [\![B]\!]_{\eta[X\mapsto\alpha]}) \right\rfloor_{k+1} && \text{(by Def. 4.3.1)}\\
&= \left\lfloor \forall_{\lfloor [\![A]\!]_\eta \rfloor_k}(\lambda\alpha.\ \left\lfloor [\![B]\!]_{\eta[X\mapsto\alpha]} \right\rfloor_k) \right\rfloor_{k+1} && \text{(by Lemma 3.6.10)}\\
&= \left\lfloor \forall_{\lfloor [\![A]\!]_{\lfloor\eta\rfloor_k} \rfloor_k}(\lambda\alpha.\ \left\lfloor [\![B]\!]_{\lfloor\eta[X\mapsto\alpha]\rfloor_k} \right\rfloor_k) \right\rfloor_{k+1} && \text{(induction hyp.)}\\
&= \left\lfloor \forall_{[\![A]\!]_{\lfloor\eta\rfloor_k}}(\lambda\alpha.\ [\![B]\!]_{\lfloor\eta[X\mapsto\alpha]\rfloor_k}) \right\rfloor_{k+1} && \text{(by Lemma 3.6.10)}\\
&= \left\lfloor [\![\underline{A}]\!]_{\lfloor\eta\rfloor_k} \right\rfloor_{k+1} && \text{(by Def. 4.3.1)}
\end{aligned}$$

Case $\underline{A} = \exists X{\leqslant}A.B$. Similarly, from Lemma 3.6.10 and the induction hypothesis.

2. We use case analysis on A in order to show $\left\lfloor [\![A]\!]_{\eta} \right\rfloor_k = \left\lfloor [\![A]\!]_{\lfloor \eta \rfloor_k} \right\rfloor_k$.

Case $A = \underline{A}$. Direct from 1. since contractiveness implies non-expansiveness.

Case $A = X$. $\left\lfloor [\![X]\!]_{\eta} \right\rfloor_{k+1} = \lfloor \eta(X) \rfloor_k = \lfloor \lfloor \eta(X) \rfloor_k \rfloor_k = \left\lfloor [\![X]\!]_{\lfloor \eta \rfloor_k} \right\rfloor_k$

Case $A = \mu X.\underline{A}$. By Def. 4.3.1 we have $[\![\mu X.\underline{A}]\!]_{\eta} = \mu(\lambda\alpha.\ [\![\underline{A}]\!]_{\eta[X \mapsto \alpha]})$. Also by the induction hypothesis $[\![\underline{A}]\!]_{\eta[X \mapsto \alpha]}$ is contractive, so we can apply Lemma 3.5.11.

$$\begin{aligned}
\left\lfloor [\![\mu X.\underline{A}]\!]_{\eta} \right\rfloor_k &= \left\lfloor \mu(\lambda\alpha.\ [\![\underline{A}]\!]_{\eta[X \mapsto \alpha]}) \right\rfloor_k && \text{(by Def. 4.3.1)} \\
&= \left\lfloor \mu(\lambda\alpha.\ \left\lfloor [\![\underline{A}]\!]_{\eta[X \mapsto \alpha]} \right\rfloor_k) \right\rfloor_k && \text{(by Lemma 3.5.11)} \\
&= \left\lfloor \mu(\lambda\alpha.\ \left\lfloor [\![\underline{A}]\!]_{\lfloor \eta[X \mapsto \alpha] \rfloor_k} \right\rfloor_k) \right\rfloor_k && \text{(induction hyp.)} \\
&= \left\lfloor \mu(\lambda\alpha.\ [\![\underline{A}]\!]_{\lfloor \eta[X \mapsto \alpha] \rfloor_k}) \right\rfloor_k && \text{(by Lemma 3.5.11)} \\
&= \left\lfloor [\![\mu X.\underline{A}]\!]_{\lfloor \eta \rfloor_k} \right\rfloor_k && \text{(by Def. 4.3.1)}
\end{aligned}$$

□

Lemma 4.3.7 (Substitution).
If $X \notin FV(B)$ *and* $(FV(A) \setminus \{X\}) \cup FV(B) \subseteq Dom(\eta)$ *then*

$$[\![A[X{:=}B]]\!]_{\eta} = [\![A]\!]_{\eta[X \mapsto [\![B]\!]_{\eta}]}$$

Proof. By induction on the structure of A.

Case $A = Top$ or $A = Bot$. Trivial.

Case $A = X$. $[\![X[X{:=}B]]\!]_{\eta} = [\![B]\!]_{\eta} = [\![X]\!]_{\eta[X \mapsto [\![B]\!]_{\eta}]}$.

Case $A = Y \neq X$. $[\![Y[X{:=}B]]\!]_{\eta} = [\![Y]\!]_{\eta} = \eta(Y) = [\![Y]\!]_{\eta[X \mapsto [\![B]\!]_{\eta}]}$.

Case $A = [m_d :_{\nu_d} A_d]_{d \in D}$.

$$\begin{aligned}
&[\![[m_d :_{\nu_d} A_d]_{d \in D}\,[X{:=}B]]\!]_{\eta} = \\
&\quad = [\![[m_d :_{\nu_d} A_d[X{:=}B]]_{d \in D}]\!]_{\eta} && \text{(by substitution)} \\
&\quad = \left[m_d :_{\nu_d} [\![A_d[X{:=}B]]\!]_{\eta}\right]_{d \in D} && \text{(by Def. 4.3.1)} \\
&\quad = \left[m_d :_{\nu_d} [\![A_d]\!]_{\eta[X \mapsto [\![B]\!]_{\eta}]}\right]_{d \in D} && \text{(by induction hyp.)} \\
&\quad = [\![[m_d :_{\nu_d} A_d]_{d \in D}]\!]_{\eta[X \mapsto [\![B]\!]_{\eta}]} && \text{(by Def. 4.3.1)}
\end{aligned}$$

Case $A = B \to B'$. Analogous to the previous case.

Case $A = \mu Y.\underline{A}$ where $Y{\neq}X$ and $Y{\notin}FV(B)$ (otherwise we α-rename Y).

$$\begin{aligned}
&\llbracket(\mu Y.\underline{A})[X{:=}B]\rrbracket_\eta = \\
&\quad= \llbracket \mu Y.(\underline{A}[X{:=}B])\rrbracket_\eta && \text{(by substitution)}\\
&\quad= \mu(\lambda\alpha.\ \llbracket \underline{A}[X{:=}B]\rrbracket_{\eta[Y\mapsto\alpha]}) && \text{(by Def. 4.3.1)}\\
&\quad= \mu(\lambda\alpha.\ \llbracket \underline{A}\rrbracket_{\eta[Y\mapsto\alpha]\left[X\mapsto\llbracket B\rrbracket_{\eta[Y\mapsto\alpha]}\right]}) && \text{(by induction hyp.)}\\
&\quad= \mu(\lambda\alpha.\ \llbracket \underline{A}\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_{\eta}\right][Y\mapsto\alpha]}) && \text{(by Property 4.3.2)}\\
&\quad= \llbracket \mu Y.\underline{A}\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_{\eta}\right]} && \text{(by Def. 4.3.1)}
\end{aligned}$$

Case $A = \forall Y{\leqslant}A'.B'$ where $Y{\neq}X$ and $Y{\notin}FV(B)$ (otherwise we α-rename Y).

$$\begin{aligned}
&\llbracket(\forall Y{\leqslant}A'.B')[X{:=}B]\rrbracket_\eta = \\
&\quad= \llbracket \forall Y{\leqslant}(A'[X{:=}B]).(B'[X{:=}B])\rrbracket_\eta && \text{(by substitution)}\\
&\quad= \forall_{\llbracket A'[X:=B]\rrbracket_\eta}\lambda\alpha.\ \llbracket B'[X{:=}B]\rrbracket_{\eta[Y\mapsto\alpha]} && \text{(by Def. 4.3.1)}\\
&\quad= \forall_{\llbracket A'\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_\eta\right]}}\lambda\alpha.\ \llbracket B'\rrbracket_{\eta[Y\mapsto\alpha]\left[X\mapsto\llbracket B\rrbracket_{\eta[Y\mapsto\alpha]}\right]} && \text{(by induction hyp.)}\\
&\quad= \forall_{\llbracket A'\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_\eta\right]}}\lambda\alpha.\ \llbracket B'\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_{\eta}\right][Y\mapsto\alpha]} && \text{(by Property 4.3.2)}\\
&\quad= \llbracket \forall Y{\leqslant}A'.B'\rrbracket_{\eta\left[X\mapsto\llbracket B\rrbracket_{\eta}\right]} && \text{(by Def. 4.3.1)}
\end{aligned}$$

Case $A = \exists Y{\leqslant}A'.B'$. Analogous to the previous case.

□

Lemma 4.3.8 (Soundness of Subtyping).
If $\Gamma \vdash A \leqslant B$ *and* $\eta \models \Gamma$, *then* $\llbracket A\rrbracket_\eta \subseteq \llbracket B\rrbracket_\eta$

Proof. By induction on the derivation of $\Gamma \vdash A \leqslant B$. Case analysis on the last applied rule.

Case (SynSubRefl) $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \leqslant A}$. $\llbracket A\rrbracket_\eta \subseteq \llbracket A\rrbracket_\eta$ immediate by SubRefl.

Case (SynSubTrans) $\dfrac{\Gamma \vdash A \leqslant A' \quad \Gamma \vdash A' \leqslant B}{\Gamma \vdash A \leqslant B}$

By the induction hypothesis we have that $\llbracket A\rrbracket_\eta \subseteq \llbracket A'\rrbracket_\eta$ and $\llbracket A'\rrbracket_\eta \subseteq \llbracket B\rrbracket_\eta$. Then by SubTrans $\llbracket A\rrbracket_\eta \subseteq \llbracket B\rrbracket_\eta$.

Case (SynSubTop) $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \leqslant \mathit{Top}}$. $\llbracket A\rrbracket_\eta \subseteq \top$ directly by SubTop.

Case (SynSubBot) $\dfrac{\Gamma \vdash A}{\Gamma \vdash \mathit{Bot} \leqslant A}$. $\bot \subseteq \llbracket A\rrbracket_\eta$ directly by SubBot.

Case (SynSubVar) $\dfrac{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash \diamond}{\Gamma_1, X{\leqslant}A, \Gamma_2 \vdash X \leqslant A}$

From $\eta \models \Gamma_1, X{\leqslant}A, \Gamma_2$ by Property 4.3.4 we also have $\eta \models \Gamma_1, X{\leqslant}A$. Definition 4.3.3 gives us then $\eta(X) \subseteq [\![A]\!]_\eta$, by Def. 4.3.1 $\eta(X) = [\![X]\!]_\eta$, so $[\![X]\!]_\eta \subseteq [\![A]\!]_\eta$.

Case (SynSubProc) $\dfrac{\Gamma \vdash A' \leqslant A \quad \Gamma \vdash B \leqslant B'}{\Gamma \vdash A{\to}B \leqslant A'{\to}B'}$

Applying the induction hypothesis to $\Gamma \vdash A' \leqslant A$ and $\Gamma \vdash B \leqslant B'$ yields $[\![A']\!]_\eta \subseteq [\![A]\!]_\eta$ and $[\![B]\!]_\eta \subseteq [\![B']\!]_\eta$ respectively. By SubProc $[\![A]\!]_\eta \to [\![B]\!]_\eta \subseteq [\![A']\!]_\eta \to [\![B']\!]_\eta$, and thus by Def. 4.3.1 we get $[\![A{\to}B]\!]_\eta \subseteq [\![A'{\to}B']\!]_\eta$.

Case (SynSubObj1) $\dfrac{\begin{array}{r}E \subseteq D \quad \forall e{\in}E.\, (\nu_e \in \{+,0\} \Rightarrow \Gamma \vdash A_e \leqslant B_e)\\ \wedge\, (\nu_e \in \{-,0\} \Rightarrow \Gamma \vdash B_e \leqslant A_e)\end{array}}{\Gamma \vdash [m_d :_{\nu_d} A_d]_{d\in D} \leqslant [m_e :_{\nu_e} B_e]_{e\in E}}$

By induction hypothesis for all $e \in E$ we get that if $\nu_e \in \{+,0\}$ then $[\![A_e]\!]_\eta \subseteq [\![B_e]\!]_\eta$, and if $\nu_e \in \{-,0\}$ then $[\![B_e]\!]_\eta \subseteq [\![A_e]\!]_\eta$. Since also $E \subseteq D$, then by VarSubObj1 we get that $\left[m_d :_{\nu_d} [\![A_d]\!]_\eta\right]_{d\in D} \subseteq \left[m_e :_{\nu_e} [\![A_e]\!]_\eta\right]_{e\in E}$, which by Def. 4.3.1 allows us to conclude that $[\![[m_d :_{\nu_d} A_d]_{d\in D}]\!]_\eta \subseteq [\![[m_e :_{\nu_e} A_e]_{e\in E}]\!]_\eta$.

Case (SynSubObj2) $\dfrac{\forall d \in D.\, \nu_d = 0 \;\vee\; \nu_d = \nu'_d}{\Gamma \vdash [m_d :_{\nu_d} A_d]_{d\in D} \leqslant [m_d :_{\nu'_d} A_d]_{d\in D}}$

By VarSubObj2 we obtain that $[m_d :_{\nu_d} [\![A_d]\!]_{\eta_d}]_{d\in D} \subseteq [m_d :_{\nu'_d} [\![A_d]\!]_{\eta_d}]_{d\in D}$, so Def. 4.3.1 gives us $[\![[m_d :_{\nu_d} A_d]_{d\in D}]\!]_\eta \subseteq \left[\!\!\left[[m_d :_{\nu'_d} A_d]_{d\in D}\right]\!\!\right]_\eta$.

Case (SynSubRec) $\dfrac{\Gamma \vdash \mu X.\underline{A} \quad \Gamma \vdash \mu Y.\underline{B} \quad \Gamma, Y{\leqslant}Top, X{\leqslant}Y \vdash \underline{A} \leqslant \underline{B}}{\Gamma \vdash \mu X.\underline{A} \leqslant \mu Y.\underline{B}}$

Assume $\eta \models \Gamma$, α and β such that $\alpha \subseteq \beta$, $\eta' \equiv \eta[Y \mapsto \beta][X \mapsto \alpha]$, $\Gamma' \equiv \Gamma, Y{\leqslant}Top, X{\leqslant}Y$. It is immediate from SubTop that $\beta \subseteq \top = [\![Top]\!]_\eta$, so $\eta[Y \mapsto \beta] \models \Gamma, Y{\leqslant}Top$. From this, because $\alpha \subseteq \beta$ and $[\![Y]\!]_{\eta[Y\mapsto\beta]} = \beta$ by Def. 4.3.3 we get that $\Gamma' \models \eta'$.

Now we can apply the induction hypothesis for $\Gamma' \vdash \underline{A} \leqslant \underline{B}$, therefore $[\![\underline{A}]\!]_{\eta'} \subseteq [\![\underline{B}]\!]_{\eta'}$. $\Gamma \vdash \mu X.\underline{A}$ implies that $Y \notin FV(\underline{A})$, and analogously $\Gamma \vdash \mu X.\underline{B}$ implies $X \notin FV(\underline{B})$, so we can reformulate $[\![\underline{A}]\!]_{\eta'} \subseteq [\![\underline{B}]\!]_{\eta'}$ to $[\![\underline{A}]\!]_{\eta[X\mapsto\alpha]} \subseteq [\![\underline{B}]\!]_{\eta[Y\mapsto\beta]}$. If we denote $F \equiv \lambda\alpha.\ [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]}$ and $G \equiv \lambda\beta.\ [\![\underline{B}]\!]_{\eta[Y\mapsto\beta]}$, then the last relation becomes $F(\alpha) \subseteq G(\beta)$. By SubRec $\mu F = \mu G$, so finally by Def. 4.3.1 we conclude that $[\![\mu X.\underline{A}]\!]_\eta \subseteq [\![\mu Y.\underline{B}]\!]_\eta$

Case (SynSubUniv) $\dfrac{\Gamma \vdash A' \leqslant A \quad \Gamma, X{\leqslant}A' \vdash B \leqslant B'}{\Gamma \vdash \forall X{\leqslant}A.B \leqslant \forall X{\leqslant}A'.B'}$

We first apply the induction hypothesis to $\Gamma \vdash A' \leqslant A$ and get that $[\![A']\!]_\eta \subseteq [\![A]\!]_\eta$, or if we denote $\alpha \equiv [\![A]\!]_\eta$ and $\beta \equiv [\![A']\!]_\eta$, then $\beta \subseteq \alpha$. Let τ be a type so that $\tau \subseteq \beta$. Def. 4.3.3 gives us that $\eta[X \mapsto \tau] \models \Gamma, X{\leqslant}A'$ so by applying the induction hypothesis again, this time for $\Gamma, X{\leqslant}A' \vdash B \leqslant B'$ we obtain that $[\![B]\!]_{\eta[X\mapsto\tau]} \subseteq [\![B']\!]_{\eta[X\mapsto\tau]}$. If we denote $F(\tau) \equiv [\![B]\!]_{\eta[X\mapsto\tau]}$ and $G(\tau) \equiv [\![B']\!]_{\eta[X\mapsto\tau]}$, then applying the rule SubUniv yields $\forall_\alpha F \subseteq \forall_\beta G$. Finally, by applying Def. 4.3.1 to both sides of the inclusion we can conclude that $[\![\forall X{\leqslant}A.B]\!]_\eta \subseteq [\![\forall X{\leqslant}A'.B']\!]_\eta$.

Case (SynSubExist) $\dfrac{\Gamma \vdash A \leqslant A' \quad \Gamma, X{\leqslant}A \vdash B \leqslant B'}{\Gamma \vdash \exists X{\leqslant}A.B \leqslant \exists X{\leqslant}A'.B'}$

Analogous to the SynSubUniv case.

□

Theorem 4.3.9 (Semantic Soundness)**.**
If $\Gamma \vdash a : A$ *and* $\eta \models \Gamma$*, then* $[\![\Gamma]\!]_\eta \models E(a) : [\![A]\!]_\eta$.

Proof. By induction on the derivation of $\Gamma \vdash a : A$ and case analysis on the last applied rule.

Case (SynSub) $\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash A \leqslant B}{\Gamma \vdash a : B}$

From $\Gamma \vdash a : A$ by the induction hypothesis we get $[\![\Gamma]\!]_\eta \models E(a) : [\![A]\!]_\eta$. Since $\Gamma \vdash A \leqslant B$ by the soundness of the subtyping relation (Lemma 4.3.8) $[\![A]\!]_\eta \subseteq [\![B]\!]_\eta$. By Sub we conclude that $[\![\Gamma]\!]_\eta \models E(a) : [\![B]\!]_\eta$.

Case (SynVar) $\dfrac{\Gamma_1, x{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, x{:}A, \Gamma_2 \vdash x : A}$

By Var $[\![\Gamma_1, x{:}A, \Gamma_2]\!]_\eta \models x : [\![\Gamma_1, x{:}A, \Gamma_2]\!]_\eta(x)$, so it suffices to show that $[\![\Gamma_1, x{:}A, \Gamma_2]\!]_\eta(x) = [\![A]\!]_\eta$. Since $\Gamma_1, x{:}A, \Gamma_2 \vdash \diamond$ we have that $x \notin \mathit{Vars}(\Gamma_2)$, so by Def. 4.3.5 $[\![\Gamma_1, x{:}A, \Gamma_2]\!]_\eta(x) = [\![\Gamma_1, x{:}X]\!]_\eta(x) = [\![A]\!]_\eta$.

Case (SynLam) $\dfrac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda x{:}A.\, b : A{\to}B}$

Since by the premises of the theorem $\eta \models \Gamma$, by Def. 4.3.3 we also have $\eta \models \Gamma, x{:}A$. So from $\Gamma, x{:}A \vdash b : B$ by the induction hypothesis $[\![\Gamma, x{:}A]\!]_\eta \models E(b) : [\![B]\!]_\eta$. By Def. 4.3.5 $[\![\Gamma, x{:}A]\!]_\eta = [\![\Gamma]\!]_\eta \left[x \mapsto [\![A]\!]_\eta\right]$. Thus by Lam $[\![\Gamma]\!]_\eta \models \lambda x.\, E(b) : [\![A]\!]_\eta \to [\![B]\!]_\eta$, and finally by the meaning of procedure types (Def. 4.3.1) we can conclude that $[\![\Gamma]\!]_\eta \models \lambda x.\, E(b) : [\![A{\to}B]\!]_\eta$.

Case (SynApp) $\dfrac{\Gamma \vdash a : B{\to}A \quad \Gamma \vdash b : B}{\Gamma \vdash a\, b : A}$

From $\Gamma \vdash a : B{\to}A$, by the induction hypothesis $[\![\Gamma]\!]_\eta \models E(a) : [\![B{\to}A]\!]_\eta$, and thus by Def. 4.3.1 $[\![\Gamma]\!]_\eta \models E(a) : [\![B]\!]_\eta \to [\![A]\!]_\eta$. Also by the induction hypothesis, from $\Gamma \vdash b : B$ we get that $[\![\Gamma]\!]_\eta \models E(b) : [\![B]\!]_\eta$. So by App we can conclude that $[\![\Gamma]\!]_\eta \models E(a)\ E(b) : [\![A]\!]_\eta$

Case (SynObj) $\dfrac{\forall d{\in}D.\ \Gamma, x{:}A \vdash b_d : A_d}{\Gamma \vdash [m_d{=}\varsigma(x_d{:}A)b_d]_{d\in D} : A}$ where $A = [m_d :_{\nu_d} A_d]_{d\in D}$

By Def. 4.3.3 from $\eta \models \Gamma$ we get $\eta \models \Gamma, x{:}A$. We chose an arbitrary d from D and we apply the induction hypothesis to $\Gamma, x{:}A \vdash b_d : A_d$ and get that $[\![\Gamma, x{:}A]\!]_\eta \models E(b_d) : [\![A_d]\!]_\eta$, or equivalently (Def. 4.3.5) that $[\![\Gamma]\!]_\eta \left[x \mapsto [\![A]\!]_\eta\right] \models E(b_d) : [\![A_d]\!]_\eta$. Now by VarObj we get that $[\![\Gamma]\!]_\eta \models [m_d{=}\varsigma(x_d)E(b_d)]_{d\in D} : \left[m_d :_{\nu_d} [\![A_d]\!]_\eta\right]_{d\in D}$, so by the meaning of object types (Def. 4.3.1) it follows that $[\![\Gamma]\!]_\eta \models [m_d{=}\varsigma(x_d)E(b_d)]_{d\in D} : [\![[m_d :_{\nu_d} A_d]_{d\in D}]\!]_\eta$.

Case (SynInv) $\dfrac{\Gamma \vdash a : A \quad e \in D \quad \nu_e \in \{+, 0\}}{\Gamma \vdash a.m_e : A_e}$ where $A = [m_d :_{\nu_d} A_d]_{d\in D}$

Similar to the previous cases.

Case (SynUpd) $\dfrac{\Gamma \vdash a : A \quad e \in D \quad \Gamma, x{:}A \vdash b : A_e \quad \nu_e \in \{-, 0\}}{\Gamma \vdash a.m_e \coloneqq \varsigma(x{:}A)b : A}$,

where $A = [m_d :_{\nu_d} A_d]_{d\in D}$. Similar to the previous cases.

Case (SynUnfold) $\dfrac{\Gamma \vdash a : \mu X.\underline{A}}{\Gamma \vdash \mathrm{unfold}_{\mu X.\underline{A}}\, a : \underline{A}[X{:=}\mu X.\underline{A}]}$

From $\Gamma \vdash a : \mu X.\underline{A}$ by the induction hypothesis we have that $[\![\Gamma]\!]_\eta \models E(a) : [\![\mu X.\underline{A}]\!]_\eta$. If we plug in the definition of $[\![\mu X.\underline{A}]\!]_\eta$, and denote $F(\alpha) \equiv [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]}$, then we equivalently get $[\![\Gamma]\!]_\eta \models E(a) : \mu F$. By Lemma 4.3.6 we can derive that F is contractive, so by Unfold $[\![\Gamma]\!]_\eta \models E(a) : F(\mu F)$. We have that $F(\mu F) \equiv [\![\underline{A}]\!]_{\eta[X\mapsto\mu(\lambda\alpha.\, [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]})]} = [\![\underline{A}]\!]_{\eta[X\mapsto[\![\mu X.\underline{A}]\!]_\eta]}$, and by the substitution lemma (Lemma 4.3.7) that $[\![\underline{A}]\!]_{\eta[X\mapsto[\![\mu X.\underline{A}]\!]_\eta]} = [\![\underline{A}[X{:=}\mu X.\underline{A}]]\!]_\eta$, so finally $[\![\Gamma]\!]_\eta \models E(a) : [\![\underline{A}[X{:=}\mu X.\underline{A}]]\!]_\eta$ as required.

Case (SynFold) $\dfrac{\Gamma \vdash a : \underline{A}[X{:=}\mu X.\underline{A}]}{\Gamma \vdash \mathrm{fold}_{\mu X.\underline{A}}\, a : \mu X.\underline{A}}$

The proof for this case is a reversed version of the proof for the Unfold case. We apply the induction hypothesis to $\Gamma \vdash a : \underline{A}[X{:=}\mu X.\underline{A}]$ and obtain that $[\![\Gamma]\!]_\eta \models E(a) : [\![\underline{A}[X{:=}\mu X.\underline{A}]]\!]_\eta$, which by Lemma 4.3.7 (substitution) gives us $[\![\Gamma]\!]_\eta \models E(a) : [\![\underline{A}]\!]_{\eta[X\mapsto\mu(\lambda\alpha.\, [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]})]}$. If we denote again $F(\alpha) \equiv [\![\underline{A}]\!]_{\eta[X\mapsto\alpha]}$, then $[\![\Gamma]\!]_\eta \models E(a) : F(\mu F)$. By Lemma 4.3.6 F is contractive, so we can apply Fold to obtain $[\![\Gamma]\!]_\eta \models E(a) : \mu F$, which finally by Def. 4.3.1 gives us $[\![\Gamma]\!]_\eta \models E(a) : [\![\mu X.\underline{A}]\!]_\eta$.

Case (SynTAbs) $\dfrac{\Gamma, X{\leqslant}A \vdash b : B}{\Gamma \vdash \Lambda X{\leqslant}A.\, b : \forall X{\leqslant}A.B}$

We need to show that $[\![\Gamma]\!]_\eta \models \Lambda.\, E(b) : [\![\forall X{\leqslant}A.B]\!]_\eta$, or equivalently by Def. 4.3.1 that $[\![\Gamma]\!]_\eta \models \Lambda.\, E(b) : \forall_{[\![A]\!]_\eta}(\lambda\alpha.\ [\![B]\!]_{\eta[X\mapsto\alpha]})$. Let us denote $\alpha \equiv [\![A]\!]_\eta$ and $F \equiv \lambda\alpha.\ [\![B]\!]_{\eta[X\mapsto\alpha]}$, so that $[\![\Gamma]\!]_\eta \models \Lambda.\, E(b) : \forall_\alpha F$ remains to be shown.

Let $\tau \subseteq \alpha$. By Def. 4.3.3 $\eta[X \mapsto \tau] \models \Gamma, X{\leqslant}A$, so we can apply the induction hypothesis for $\Gamma, X{\leqslant}A \vdash b : B$, and get that $[\![\Gamma, X{\leqslant}A]\!]_{\eta[X\mapsto\tau]} \models E(b) : [\![B]\!]_{\eta[X\mapsto\tau]} \equiv F(\tau)$. By Def. 4.3.5 and the fact that $X \notin \mathit{TypeVars}(\Gamma)$ we have $[\![\Gamma, X{\leqslant}A]\!]_{\eta[X\mapsto\tau]} \triangleq [\![\Gamma]\!]_{\eta[X\mapsto\tau]} = [\![\Gamma]\!]_{\eta}$. Putting them together, we obtain $[\![\Gamma]\!]_{\eta} \models E(b) : F(\tau)$, which by TAbs gives us $[\![\Gamma]\!]_{\eta} \models E(b) : \forall_\alpha F$.

Case (SynTApp) $\dfrac{\Gamma \vdash a : \forall X{\leqslant}A.B \quad \Gamma \vdash A' \leqslant A}{\Gamma \vdash a\ A' : B[X{:=}A']}$

We apply the induction hypothesis to $\Gamma \vdash a : \forall X{\leqslant}A.B$ and obtain that $[\![\Gamma]\!]_{\eta} \models E(a) : [\![\forall X{\leqslant}A.B]\!]_{\eta}$. If we denote $\alpha \equiv [\![A]\!]_{\eta}$ and $F(\alpha) \equiv [\![B]\!]_{\eta[X\mapsto\alpha]}$, then by the meaning of universal types $[\![\Gamma]\!]_{\eta} \models E(a) : \forall_\alpha F$. From $\Gamma \vdash A' \leqslant A$ by the soundness of subtyping (Lemma 4.3.8) we have that $[\![A']\!]_{\eta} \subseteq [\![A]\!]_{\eta} \equiv \alpha$. By TApp $[\![\Gamma]\!]_{\eta} \models E(a) : F([\![A']\!]_{\eta})$, and from this by Lemma 4.3.7 (substitution) we finally obtain that $[\![\Gamma]\!]_{\eta} \models E(a) : [\![B[X{:=}A']]\!]_{\eta}$.

Case (SynPack) $\dfrac{\Gamma \vdash C \leqslant A \quad \Gamma \vdash a[X{:=}C] : B[X{:=}C]}{\Gamma \vdash \text{pack } X{\leqslant}A = C \text{ in } a{:}B : \exists X{\leqslant}A.B}$

We need to show that $[\![\Gamma]\!]_{\eta} \models \Lambda.\, E(b) : [\![\exists X{\leqslant}A.B]\!]_{\eta}$, or equivalently by Def. 4.3.1 that $[\![\Gamma]\!]_{\eta} \models \Lambda.\, E(b) : \exists_{[\![A]\!]_{\eta}}(\lambda\alpha.\ [\![B]\!]_{\eta[X\mapsto\alpha]})$. As in the previous cases, we denote $\alpha \equiv [\![A]\!]_{\eta}$ and $F \equiv \lambda\alpha.\ [\![B]\!]_{\eta[X\mapsto\alpha]}$. Thus it remains to be shown that $[\![\Gamma]\!]_{\eta} \models \Lambda.\, E(b) : \exists_\alpha F$.

By Lemma 4.3.8 (soundness of subtyping) applied to $\Gamma \vdash C \leqslant A$ we get that $[\![C]\!]_{\eta} \subseteq [\![A]\!]_{\eta} \equiv \alpha$. From $\Gamma \vdash a[X{:=}C] : B[X{:=}C]$ by the induction hypothesis we get that $[\![\Gamma]\!]_{\eta} \models \Lambda.\, E(b) : [\![B[X{:=}C]]\!]_{\eta}$. By the substitution lemma (Lemma 4.3.7) we have $[\![B[X{:=}C]]\!]_{\eta} = [\![B]\!]_{\eta[X\mapsto[\![C]\!]_{\eta}]} \equiv F([\![C]\!]_{\eta})$. Finally, by Pack we have that $[\![\Gamma]\!]_{\eta} \models \Lambda.\, E(b) : \exists_\alpha F$.

Case (SynOpen) $\dfrac{\Gamma \vdash a : \exists X{\leqslant}A.B \quad \Gamma \vdash D \quad \Gamma, X{\leqslant}A, x{:}B \vdash b : D}{\Gamma \vdash \text{open } a \text{ as } X{\leqslant}A, x{:}B \text{ in } b{:}D : D}$

From $\Gamma \vdash a : \exists X{\leqslant}A.B$ by the induction hypothesis we get $[\![\Gamma]\!]_{\eta} \models E(a) : [\![\exists X{\leqslant}A.B]\!]_{\eta}$. If we denote $\alpha \equiv [\![A]\!]_{\eta}$ and $F(\alpha) \equiv [\![B]\!]_{\eta[X\mapsto\alpha]}$, then by Def. 4.3.1 $[\![\Gamma]\!]_{\eta} \models E(a) : \exists_\alpha F$.

Let τ be a type such that $\tau \subseteq \alpha$. By Def. 4.3.3 $\eta[X \mapsto \tau] \models \Gamma, X{\leqslant}A, x{:}B$, so we can apply the induction hypothesis to $\Gamma, X{\leqslant}A, x{:}B \vdash b : D$ and get that $[\![\Gamma, X{\leqslant}A, x{:}B]\!]_{\eta[X\mapsto\tau]} \models E(b) : [\![D]\!]_{\eta[X\mapsto\tau]}$. By Def. 4.3.5 and the fact that $X \notin \mathit{TypeVars}(\Gamma)$ we get $[\![\Gamma, X{\leqslant}A, x{:}B]\!]_{\eta[X\mapsto\tau]} \triangleq [\![\Gamma, X{\leqslant}A]\!]_{\eta[X\mapsto\tau]} \left[x \mapsto [\![B]\!]_{\eta[X\mapsto\tau]}\right] = [\![\Gamma]\!]_{\eta}[x \mapsto F(\tau)]$, thus $[\![\Gamma]\!]_{\eta}[x \mapsto F(\tau)] \models E(b) : [\![D]\!]_{\eta}$.

Finally, by Open since $[\![\Gamma]\!]_{\eta} \models E(a) : \exists_\alpha F$ and since for all $\tau \subseteq \alpha$ we have $[\![\Gamma]\!]_{\eta}[x \mapsto F(\tau)] \models E(b) : [\![D]\!]_{\eta}$, we conclude that $[\![\Gamma]\!]_{\eta} \models \text{open } E(a) \text{ as } x \text{ in } E(b) : [\![A]\!]_{\eta}$.

□

Corollary 4.3.10 (Type Safety). *Well-typed terms evaluate safely once erased.*

Proof. Immediate from Theorems 4.3.9 and 3.1.12. □

Chapter 5

Conclusion

5.1 Summary

In this thesis we introduced a step-indexed semantic model for the functional object calculus, and used it to prove the soundness of an expressive type system with object types, subtyping, recursive and bounded quantified types.

We started by presenting the functional object calculus, a very simple object-oriented language. Its only primitives are the objects, collections of methods that can be invoked and updated. The methods of an object can call each other through a self argument, and this permits direct recursion. We showed how other important constructs including procedures and classes can be encoded using objects.

Using only the small-step operational semantics of an untyped calculus we constructed purely semantic types as sets of values indexed by computation steps. Intuitively, a term has a certain semantic type if it behaves like an element of that type for any number of computation steps. The semantic types were built as sequences of increasingly accurate semantic approximations, and this was crucial when considering the term-level recursion inherent to objects, but also when considering recursive types. The semantic type judgement was defined independently of any typing rules and immediately guarantees the safety of typable terms. The semantic typing rules are just lemmas that we proved sound with respect to the model.

An important aspect of this work is the study of subtyping in a step-indexed model, and its interaction with object types, recursive and quantified types. The most natural definition of object types as collections of well-typed methods does not validate any notion of subtyping because of the contravariance of method types. So we first refined the definition of object types to validate subtyping in width, then by restricting certain invocations and updates we could soundly allow subtyping in depth for object types extended with variance annotations.

Type checking terms using the semantic typing rules is undecidable, so we introduced a syntactic type system for which type checking can be made decidable. Finally, we proved the soundness of the syntactic type system with respect to the semantic model, which immediately implies that well-typed terms are safe to evaluate after type erasure.

5.2 Related Work

The functional object calculus was introduced by Abadi and Cardelli for investigating the theoretical properties of objects [AC96a]. The untyped ς-calculus and the syntactic type system we presented here are their work [AC96b, AC95b]. Abadi and Cardelli prove the soundness of their second-order equational theories by constructing a fairly intricate denotational semantic model based on metric spaces [MPS86]. The soundness of their second-order type system is an immediate consequence of the soundness of the equational theory [AC96a, Chapter 14]. If one was only concerned with types and not with equations, then a subject-reduction proof would suffice. Abadi and Cardelli give subject-reduction proofs for all the other type systems in their book.

Step-indexed semantic models were introduced by Appel and his collaborators in the context of foundational proof-carrying code [AF00]. Their goal was to construct more elementary and more modular proofs of type soundness that can be easier checked automatically. They were primarily interested in low-level languages, however they also applied their technique to a pure λ-calculus with recursive types [AM01]. Later they successfully extended it to general references, impredicative polymorphism [AAV03, Ahm04] and substructural state [AFM05]. The step-indexed semantic model we presented extends the one for recursive types of Appel and McAllester [AM01] with method and object types, subtyping and bounded quantified types.

Subtyping in a step-indexed semantic model was previously considered by Swadi who studied a typed machine language [Swa03, Section 3.4]. Our setup is however much different than the one of Swadi, so the subtle issues concerning object types are original to our work. Swadi was also the first to explicitly consider type variables in a step-indexed semantic model and used a semantic kind system to track the contractiveness and non-expansiveness of types with free type variables [ARS02, Swa03]. We avoid having a more complex model with type variables by additionally considering a syntactic type system which we prove sound with respect to the semantic model.

Proving soundness with respect to denotational semantic models dates back to the 70's and was quite popular until it was replaced by subject-reduction. The paper on subject-reduction of Wright and Felleisen gives a survey on the type safety proofs based on denotational semantics and discusses some of their problems [WF94, Section 2]. Most important, such proofs rely on different and often unrelated proof techniques, and even a minor extension to a language may require a complete restructuring of its denotational semantics and so a completely new approach to re-establish soundness. The monadic λ-calculi [Mog91] or the call-by-push-value calculus [Lev04] aim at structuring the denotational semantics in such a way that at least some extensions can be done easier, still they give no complete solution to such problems. Soundness proofs based on step-indexed semantic models tend to be less affected by extensions to the underlying calculus than proofs based on denotational models, and it seems that at least in some cases they are more modular than subject-reduction proofs.

5.3 Future Work

Step-indexed Model for the Imperative Object Calculus

The results of this thesis can be extended to the imperative object calculus of Abadi and Cardelli [AC95a] [HS07]. Syntactically the imperative object calculus is very similar to the functional calculus considered in this thesis. However, its semantics is defined with respect to a store: method invocations read from the store and method updates modify the store. Executable code is also stored (i.e. the store is higher-order) and aliasing is possible.

The semantic model for the imperative object calculus is an extension of the step-indexed model for general references of Ahmed et. al. [AAV03, Ahm04]. We refine the general references from Ahmed's model to also encompass read-only and write-only references, in a similar way to the references in the Forsythe programming language [Rey88, Rey96] [Pie02, Section 15.5]. We also add object types with variance annotations, subtyping and bounded quantifiers to Ahmed's model. And although in the resulting model the semantic types are more complex than the ones from this thesis, the model satisfies exactly the same semantic typing rules (i.e. the interface given by Figures 3.5 and 3.4).

Even more, the syntactic type system for the imperative calculus is exactly the same as the one in this thesis, so all the results in Chapter 4, in particular the semantic soundness theorem (Theorem 4.3.9) and the type safety corollary (Corollary 4.3.10), directly apply to the imperative object calculus. Well-typed terms do not get stuck no matter whether they are evaluated in a functional or an imperative way[1]. It would not be possible to prove such a result using subject-reduction, since the typing judgement for the imperative calculus would also depend on a "store typing" that maps store locations to types, and thus be different from the judgement for the functional calculus. However, since we are not using subject-reduction, we do not need to type-check partially evaluated terms which contain locations.

Very Modal Models of Types for Imperative Objects

The step-indexed model of Appel and McAllester [AM01] used in this thesis, as well as the model for general references of Ahmed et. al. [AAV03, Ahm04] which we extended and used for the imperative object calculus share a common problem. The operations on values of quantified types require taking steps in the operational semantics which are essentially no-ops. Ideally in an untyped setting we expect to have no introduction and elimination forms for quantified types such as pack and unpack (see Section 3.6.1). More severely, in the case of the imperative object calculus the operational semantics of method invocation had to be modified to take two computation steps, in order for it to match the two different operations happening in the model: dereferencing a cell to get a procedure and then applying the procedure to the self argument.

Appel et. al. recently proposed a new semantic model which among other improvements also addresses this problem [AMRV07]. In this new model op-

[1]This supports the claim of Abadi and Cardelli that the evaluation model is not important when typing object calculi [AC96a]. It also shows that a very important feature such as state can be added to a language without causing major problems for its type soundness proofs based on step-indexed models.

erations such as type abstraction and application are just coercions without operational significance, and there are no explicit introduction and elimination forms for quantified types. It would be thus interesting to see whether we can more naturally accommodate imperative objects in this more general model.

Binary Logical Relations for Functional Objects

Proving the equivalence of programs is needed for verifying the correctness of program transformations, for showing that the behavior of programs does not depend on the implementation of abstract data types, for assuring that secret information is not leaked by a program and more. In most settings we are interested in proving that two programs are contextually equivalent, i.e. they have the same observable behaviour in any context. However, this involves quantification over all contexts so it is difficult to show directly, and one usually employs a method known as logical relations [Tai67, Gir72, Plo73, Pit98, Pit00].

Appel and McAllester gave not only a step-indexed model of recursive types, but also sketched a step-indexed partial equivalence relation model [AM01]. Ahmed later modified this model to obtain a step-indexed binary logical relation that she proved to be an equivalence (the transitivity of the original relation is still open). She also extended the model to impredicative quantified types and showed that her relation captures exactly contextual equivalence [Ahm06]. This was all done in a pure functional setting, and Ahmed hopes to scale it up to support dynamically allocated mutable references. Equational reasoning in the presence of state is generally regarded as a hard problem, one that is still open.

In the context of object calculi it would be interesting to see whether data abstraction is preserved in the presence of objects, or if subsumption for objects enforces information hiding. For the functional object calculus this should be possible by extending Ahmed's logical relation to object types and subtyping, using probably some of the results of this thesis.

Program Logic for the Imperative Object Calculus

The imperative object calculus features dynamically-allocated higher-order store (i.e. executable code can be dynamically stored on the heap). This feature is present in different forms in most practical programming languages: pointers to functions in C, callbacks in Java, or general references in ML.

Purely syntactic arguments do not suffice for proving the soundness of a program logic for the imperative object calculus. The meaning of assertions is not obvious, since they have to describe the code on the heap. In this setting one can only prove soundness with respect to a semantic model, which makes a clear distinction between validity and derivability: a logic is sound if everything that can be derived using the rules is also valid with respect to the model.

However, finding good semantic models in which one can reason about the behaviour of programs is challenging for languages with dynamically-allocated higher-order store. Classical denotational models based on ordered sets tend to become very complex. Usually for modelling dynamic allocation alone one has to move to a possible-worlds model, formalized as a category of functors over complete partial orders. And while the denotational models achieve the goal of separating the notion of logical validity from derivability, in the existing models many operationally valid equations involving state do not hold.

The only program logic for the imperative object calculus found in the literature is the logic of objects of Abadi and Leino, which extends a simple type system with first-order logic formulas about the store [AL97, AL04]. Reus and Schwinghammer proved the soundness of this logic with respect to a denotational model and extended it to recursive specifications [RS06a, Sch06]. This denotational model is already quite complex, still it is not abstract enough to capture many natural equivalences involving state, and it does not have an elegant treatment of subtyping.

Therefore it would be interesting to investigate whether the techniques used for building simpler semantic models such as step-indexing, can also be used to prove the soundness of the logic of objects. For example Benton used step-indexing as a technical device together with a notion of orthogonality relating expressions to contexts to show the soundness of a compositional program logic for a very simple stack-based abstract machine [Ben05]. He also employed step-indexing in a Floyd-Hoare-style framework based on relational parametricity for the specification and verification of machine code programs [Ben06].

The logic of Abadi and Leino is similar to Floyd-Hoare logic [Flo67, Hoa69] and it does not allow local or modular reasoning [PB05, KBAR06, NAMB07]. It would thus be interesting to design a better program logic for the imperative object calculus, one that allows local reasoning about the store like the separation logic of O'Hearn and Reynolds [ORY01, Rey02]. The first separation logic for higher-order store was recently proposed by Reus and Schwinghammer, for a very simple language of commands, so objects would be a big step forward [RS06b]. Finally, maybe the new logic could also allow modular reasoning about abstract data structures, similarly to Reynolds' specification logic [Rey82].

Appendix A

Technical Definitions

A.1 Free Variables

$FV(x) \triangleq \{x\}$

$FV(\varsigma(x)b) \triangleq FV(b) \setminus \{x\}$

$FV([m_d{=}\varsigma(x_d)b_d]_{d\in D}) \triangleq \bigcup_{d\in D} FV(\varsigma(x_d)b_d)$

$FV(a.m) \triangleq FV(a)$

$FV(a.m \coloneqq \varsigma(x)b) \triangleq FV(a) \cup FV(\varsigma(x)b)$

$FV(\Lambda.\, a) \triangleq FV(a)$

$FV(a\,[]) \triangleq FV(a)$

$FV(\text{pack } a) \triangleq FV(a)$

$FV(\text{open } a \text{ as } x \text{ in } b) \triangleq FV(a) \cup (FV(b) \setminus \{x\})$

A.2 Capture-avoiding substitution

Let σ be a finite map from variables to terms (a substitution). Then we define the result of applying σ as follows.

$\sigma(x) \triangleq \sigma(x)$[1], if $x{\in}Dom(\sigma)$

$\sigma(x) \triangleq x$, if $x{\notin}Dom(\sigma)$

$\sigma(\varsigma(x)b) \triangleq \varsigma(x')(\sigma\,[x'\!\uparrow]\,([x \mapsto x']\,(b)))$[2][3], if $x'{\notin}FV(\varsigma(x)b){\cup}\bigcup_{x\in Dom(\sigma)} FV(\sigma(x))$

$\sigma([m_d{=}\varsigma(x_d)b_d]_{d\in D}) \triangleq [m_d{=}\sigma(\varsigma(x_d)b_d)]_{d\in D}$

[1]Our notation is the same for applying a substitution to a term (the left-hand side), and applying a substitution to a variable (the right-hand side). The first is the homomorphic extension of the second.

[2]By $\sigma\,[x\!\uparrow]$ we denote σ where the binding for x is removed: $\sigma\,[x\!\uparrow] \triangleq \sigma \setminus \{(x, \sigma(x))\}$.

[3]By $[x \mapsto y]$ we denote a singleton map which only maps x to y: $[x \mapsto y] \triangleq \{(x, y)\}$

$\sigma(a.m) \triangleq (\sigma(a)).m$

$\sigma(a.m \coloneqq \varsigma(x)b) \triangleq \sigma(a).m \coloneqq \sigma(\varsigma(x)b)$

$\sigma(\Lambda.\, a) \triangleq \Lambda.\, \sigma(a)$

$\sigma(a\,[]) \triangleq \sigma(a)\,[]$

$\sigma(\text{pack } a) \triangleq \text{pack } \sigma(a)$

$\sigma(\text{open } a \text{ as } x \text{ in } b) \triangleq \text{open } \sigma(a) \text{ as } x' \text{ in } (\sigma\,[x'\uparrow]\,([x \mapsto x']\,(b)))$,
if $x' \notin FV(\varsigma(x)b) \cup \bigcup_{x\in Dom(\sigma)} FV(\sigma(x))$

Special Case: Closed Substitution

We say that a substitution σ is closed if $\forall x{\in}Dom(\sigma).\ FV(\sigma(x)) = \emptyset$. If σ is closed, then in the previous definition we can always trivially satisfy the side-condition on the fresh variables by chosing $x' = x$. This gives us a simpler, deterministic way of applying closed substitutions that does not employ α-renaming. This case is particularly interesting, since our value environments are closed substitutions.

$\sigma(x) \triangleq \sigma(x)$, if $x{\in}Dom(\sigma)$

$\sigma(x) \triangleq x$, if $x{\notin}Dom(\sigma)$

$\sigma([m_d{=}\varsigma(x_d)b_d]_{d\in D}) \triangleq [m_d{=}\varsigma(x_d)(\sigma\,[x_d\uparrow]\,(b_d))]_{d\in D}$

$\sigma(a.m) \triangleq (\sigma(a)).m$

$\sigma(a.m \coloneqq \varsigma(x)b) \triangleq (\sigma(a)).m \coloneqq \varsigma(x)(\sigma\,[x\uparrow]\,(b))$

$\sigma(\Lambda.\, a) \triangleq \Lambda.\, \sigma(a)$

$\sigma(a\,[]) \triangleq \sigma(a)\,[]$

$\sigma(\text{pack } a) \triangleq \text{pack } \sigma(a)$

$\sigma(\text{open } a \text{ as } x \text{ in } b) \triangleq \text{open } \sigma(a) \text{ as } x \text{ in } \sigma\,[x\uparrow]\,(b)$

A.3 Type erasure

$E(x) = x$

$E([m_d{=}\varsigma(x_d{:}A)b_d]_{d\in D}) = [m_d{=}\varsigma(x_d)b_d]_{d\in D}$

$E(a.m_e) = E(a).m_e$

$E(a.m_e \coloneqq \varsigma(x{:}A)b) = E(a).m_e \coloneqq \varsigma(x)E(b)$

$E(\lambda x{:}A.\, b) = \lambda x.\, E(b)$

$E(a\ b) = E(a)\ E(b)$

$E(\text{fold}_{\mu X.\underline{A}}\, a) = E(a)$

$E(\mathrm{unfold}_{\mu X.\underline{A}}\, a) = E(a)$

$E(\Lambda X{\leqslant}A.\, b) = \Lambda.\, E(b)$

$E(a\ A') = E(a)\,[]$

$E(\mathrm{pack}\ X{\leqslant}A = C\ \mathrm{in}\ a\,{:}B) = \mathrm{pack}\ E(A)$

$E(\mathrm{open}\ a\ \mathrm{as}\ X{\leqslant}A, x{:}B\ \mathrm{in}\ b\,{:}D) = \mathrm{open}\ E(a)\ \mathrm{as}\ x\ \mathrm{in}\ E(b)$

A.4 Free Type Variables

$FV(\mathit{Top}) = FV(\mathit{Bot}) = \emptyset$

$FV(X) = \{X\}$

$FV(\mu X.\underline{A}) = FV(\underline{A}) \setminus \{X\}$

$FV(A{\to}B) = FV(A) \cup FV(B)$

$FV([m_d :_{\nu_d} A_d]_{d\in D}) = \bigcup_{d\in D} FV(A_d)$

$FV(\forall X{\leqslant}A.B) = FV(A) \cup FV(B) \setminus \{X\}$

$FV(\exists X{\leqslant}A.B) = FV(A) \cup FV(B) \setminus \{X\}$

A.5 Type Substitution

We consider types to be syntactically equal up to the α-renaming of bound type variables, so that the side conditions for the last three cases in the definition can always be satisfied by α-renaming type variables.

$X[X{:=}B] = A$

$Y[X{:=}B] = Y$, if $X \neq Y$

$\mathit{Top}[X{:=}B] = \mathit{Top}$

$\mathit{Bot}[X{:=}B] = \mathit{Bot}$

$(A{\to}A')[X{:=}B] = (A[X{:=}B]){\to}(A'[X{:=}B])$

$[m_d :_{\nu_d} A_d]_{d\in D}\,[X{:=}B] = [m_d :_{\nu_d} A_d[X{:=}B]]_{d\in D}$

$(\mu Y.\underline{A})[X{:=}B] = \mu Y.(\underline{A}[X{:=}B])$, if $Y{\neq}X$ and $Y{\notin}FV(B)$

$(\forall Y{\leqslant}A'.B')[X{:=}B] = \forall Y{\leqslant}(A'[X{:=}B]).(B'[X{:=}B])$, if $Y{\neq}X$ and $Y{\notin}FV(B)$

$(\exists Y{\leqslant}A'.B')[X{:=}B] = \exists Y{\leqslant}(A'[X{:=}B]).(B'[X{:=}B])$, if $Y{\neq}X$ and $Y{\notin}FV(B)$

Bibliography

[AAV03] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Princeton University, January 2003.

[AC95a] Martín Abadi and Luca Cardelli. An imperative object calculus. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[AC95b] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995.

[AC96a] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AC96b] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

[AF00] Andrew W. Appel and Amy P. Felty. A Semantic Model of Types and Machine Instructions for Proof-Carrying Code. In *POPL 2000: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, Boston, MA, Jan 2000. ACM Press.

[AFM05] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In Olivier Danvy and Benjamin C. Pierce, editors, *International Conference on Functional Programming (ICFP'05)*, pages 78–91. ACM Press, 2005.

[Ahm04] Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[Ahm06] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Proceedings of Theory and Practice of Software Development*, volume 1214 of

Lecture Notes in Computer Science, pages 682–696. Springer, 1997. Superseded by [AL04].

[AL04] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, Lecture Notes in Computer Science, pages 11–41. Springer, 2004.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007.

[ARS02] Andrew Appel, Christopher Richards, and Kedar Swadi. A kind system for typed machine language. Technical report, Princeton University, September 2002.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[Ben05] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In Zoltan Esik, editor, *Asian Symposium on Programming Languages and Systems APLAS'05*, volume 3780 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2005.

[Ben06] Nick Benton. Abstracting allocation: the new new thing. In *Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[Car85] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Springer, 1985.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[FHM94] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In Jacob T. Schwartz, editor, *Proceedings of Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, April 1967.

[Gir72] J.-Y. Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur*. These de doctorat d'tat, Universit Paris 7, 1972.

[GR96] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd Symposium on Principles of Programming Languages*, pages 386–395, January 1996.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[HS07] Cătălin Hriţcu and Jan Schwinghammer. Step-indexed semantic model of types for the imperative object calculus. Unpublished manuscript, May 2007.

[JT93] Achim Jung and Jerzy Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculus and Applications*, pages 245–257, 1993.

[KBAR06] Neelakantan R. Krishnaswami, Lars Birkedal, Jonathan Aldrich, and John C. Reynolds. Idealized ML and its separation logic. Submitted, 2006.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[Lev04] Paul Blain Levy. *Call-By-Push-Value. A Functional/Imperative Synthesis*, volume 2 of *Semantic Structures in Computation*. Kluwer, 2004.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[MPS86] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, October 1986.

[MTM96] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1996.

[NAMB07] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. 2007. To Appear in ESOP 2007.

[OR95] Peter W. O'Hearn and Jon G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, 1995.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[PB05] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 247–258. ACM, 2005.

[Pie92] Benjamin C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, 1992.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pit98] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 1998.

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

[Plo73] Gordon D. Plotkin. Lambda-definability and logical relations, 1973. Memorandum SAI-RM-4, University of Edinburgh.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[Rey82] John C. Reynolds. Idealized Algol and its specification logic. In Danielle Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[Rey88] John C. Reynolds. Preliminary design of the programming language forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, 1988. Superseded by [Rey96].

[Rey96] John C. Reynolds. Design of the programming language forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173-233, Birkhäuser, 1997.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[RS06a] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[RS06b] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 575–590. Springer, 2006.

[Sch06] Jan Schwinghammer. *Reasoning about Denotations of Recursive Objects*. PhD thesis, Department of Informatics, University of Sussex, 2006.

[Swa03] Kedar N. Swadi. *Typed Machine Language*. PhD thesis, Princeton University, July 2003.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.