A Theory of Primitive Objects

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We investigate calculi that support *method override* in presence of *object subsumption*. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the ability to modify the behavior of an object, or class, by replacing one of its methods (and *inheriting* the other methods).

At the programming-language level, all common object-oriented languages allow some combination of subsumption and override, and most handle it correctly. However, type correctness is often achieved via rather subtle conditions. We hope to illuminate the origin of some of these conditions, and illustrate the pitfalls that they avoid.

At the semantic and type-theoretic level, where one aims for generality, the combination of subsumption and override has proven difficult to model. We provide a simple calculus that supports those features.

We begin the paper with a challenge: finding an adequate type system for an untyped object calculus. We think the challenge is interesting because, first of all, the calculus is patently object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. Second, the calculus is very simple, with just four syntactic forms, and even without functions. Finally, the calculus is expressive: it can encode the λ-calculus, and can express object-flavored examples in a direct way.

We give four untyped examples. One is the classical geometric-point example. The next two (objects with backup, natural numbers as objects) are particularly intriguing, and are intended to show the flexibility of the calculus. The last one is a small programming example.

After describing the untyped calculus, we define type systems, equational theories, and denotational semantics. We account for subsumption, field update, method override, and the so-called Self type. The final formal system is obtained as a conventional second-order extension of a rather elementary first-order theory of objects. We provide typed versions of the four examples.

The typing and equational properties of objects are axiomatized, not defined. Their soundness is guaranteed by a denotational semantics. This semantics suggests a syntactic translation from our calculus into a simpler calculus with neither subtyping nor objects.

Contents

1. Introduction 4
   1.1 Primitive Semantics of Objects 5
   1.2 Derived Semantics 6
   1.3 Denotational Semantics 8
   1.4 Paper Outline 9
2. Untyped Calculi 10
   2.1 The Untyped ς-calculus 10
   2.2 Functions as Objects 12
   2.3 Fixpoints 13
   2.4 Four Examples 14
      2.4.1 Movable Points 14
      2.4.2 Backup Methods 15
      2.4.3 Object-Oriented Natural Numbers 15
      2.4.4 A Calculator 16
3. First-Order Calculi 17
   3.1 The Object Fragment 17
   3.2 A Complete Calculus 18
      3.2.1 Unique Types 19
   3.3 Equational Theory of Ob_1 20
   3.4 Functions as Objects 21
4. First-Order Calculi with Subsumption 23
   4.1 A Calculus with Subtyping 23
      4.1.1 Minimum Types 24
   4.2 Programming in $Ob_{1<:}$ 26
   4.3 Classes and Inheritance 26
      4.3.1 Classes as Collections of Pre-Methods 26
      4.3.2 Programming with Classes 27
   4.4 Equational Theory of $Ob_{1<:}$ 29
   4.5 Objects Versus Records 30
      4.5.1 Covariant Object Types 30
      4.5.2 Method Extraction 31
      4.5.3 Elder 31
5. First-Order Calculi with Recursion 32
   5.1 Recursion 32
   5.2 Recursion and Subsumption 34
   5.3 Typing the Examples 35
   5.4 The Shortcomings of $Ob_{1<:\mu}$ 36
   5.5 Towards the Type Self 37
6. Second-Order Calculi 38
   6.1 The Universal Quantifier 38
   6.2 The Existential Quantifier 40
   6.3 An Encoding of Variant Product and Function Types 42
   6.4 An Encoding of Variant Object Types 43
   6.5 The Self Quantifier 44
   6.6 A Pure Second-Order Object Calculus 47
7. Object Types with Self 47
   7.1 ς-Objects 47
   7.2 Four Examples, with Typing 51
      7.2.1 Movable Points 51
      7.2.2 Backup Methods 52
      7.2.3 Object-Oriented Natural Numbers 53

7.2.4 A Calculator 53
7.3 Classes and Inheritance, with Self 54
8. Overriding and Self 54
8.1 Overriding from the Outside 55
8.2 Recoup 56
8.2.1 The Recoup Method 56
8.2.2 The Recoup Invariant 58
8.2.3 Recoup as the Only True Method 58
8.3 Objects with Structural Invariants 59
8.3.1 Rules with Structural Side Conditions 60
8.3.2 Rules with Structural Assumptions 64
9. A Semantics for Objects 66
9.1 The Untyped Universe 66
9.2 The Interpretation of Untyped Terms 67
9.3 Types in the Untyped Universe 68
9.3.1 Preliminaries 68
9.3.2 Semantic Definitions 68
9.3.3 Lemmas about Unions 69
9.3.4 Metric Properties 73
9.3.5 Semantic Properties of Objects 75
9.4 The Interpretation of Types and Typed Terms 77
9.4.1 Interpreting Typed Terms 77
9.4.2 Interpreting Types 77
9.4.3 Verification of Selected Rules 78
10. A Translation and its Coherence 84
10.1 The Target Calculus 84
10.2 The Translation of $Ob_{1<:}$ 85
10.3 Coherence for $Ob_{1<:}$ 87
11. Conclusions 88
11.1 Related Work 89
11.2 Acknowledgments 90
Appendix A: Simple-Objects Fragments 91
Appendix B: ς-Objects Fragments (Derived Rules) 92
Appendix C: The $F_{<:\mu}$ Calculus 93
Appendix D: The ςOb Calculus 96
Appendix E: Formal Contractiveness for $FOb_{<:\mu}$ 98
Appendix F: The Target Calculus 99
References 103

1. Introduction

Typed λ-calculi have provided a rich foundation for procedural languages, but attempts to use these calculi to model object-oriented languages have not been completely successful. Instead of struggling with complex encodings of objects as λ-terms, we take objects as primitive and concentrate on the rules that they should obey. We aim to study the intrinsic properties of objects by developing object calculi that are as simple and fruitful as λ-calculi.

We investigate calculi that support *method override* in presence of *object subsumption*. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the operation that modifies the behavior of an object, or class, by replacing one of its methods; the other methods are *inherited*.

All common object-oriented languages allow some combination of subsumption and override, and most handle it correctly. However, type correctness is often achieved via rather subtle conditions. We hope to illuminate the origin of some of these conditions, and illustrate the pitfalls that they avoid.

At the semantic and type-theoretic level, where one aims for generality, the combination of subsumption and override has proven difficult to model; see section 11.1 for a discussion of previous work. We provide simple calculi that support those features.

We study calculi that include objects with methods for components, method invocation, and method override. A method is a function having a special parameter, often called *self*, that denotes the same object the method belongs to. A *field* is a degenerate method that does not make use of its self parameter. Note that it is sometimes desirable to override a field with a proper method, transparently converting passive data into active computation.

We begin this paper with a challenge: finding an adequate type system for an untyped object calculus. We think the challenge is interesting because, first of all, the calculus is patently object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. Second, the calculus is very simple, with just four syntactic forms, and even without functions. Finally, the calculus is expressive: it can encode the untyped λ-calculus, and can express object-flavored examples in a direct way.

We give four untyped examples. One is the classical geometric-point example. The next two (objects with backup, natural numbers as objects) are particularly intriguing, and are intended to show the flexibility of the calculus. The last one is a small programming example.

After describing the untyped calculus, we define type systems, equational theories, and denotational semantics. We account for subsumption, field update, method override, and the so-called Self type (which is the type of self). The final formal system is obtained as a conventional second-order extension of a rather elementary first-order theory of objects; its power comes from the interaction of these two features. We provide typed versions of the four examples.

The typing and equational properties of objects are axiomatized, not defined. Their soundness is guaranteed by a denotational semantics. We construct a model based on partial equivalence relations (pers). In our interpretation, objects are records of functions, object types are certain unions of recursive record types, and subtypes are subsets. Along the way, we study unions of pers, and thereby obtain a per semantics for abstract data types and partially abstract data types. We prove the soundness of both typing and equational rules.

The per interpretation is direct enough to be informative. In particular, it suggests a syntactic translation from our calculus to a less unconventional calculus, with neither subtyping nor objects.

1.1 Primitive Semantics of Objects

We start by investigating the operational semantics of an untyped calculus of objects, while keeping in mind some future typing requirements. Our goal in this section is to define a direct semantics of objects, viewing them as primitive.

We consider a kernel language including *object formation*, *method invocation*, and *method override*. A *field* is a degenerate method that ignores its self parameter; we talk about *field selection* and *field update*. We use the terms selection and invocation and the terms update and override somewhat interchangeably.

To avoid premature commitments, we avoid any explicit encoding of objects in terms of λ-abstraction and application. In particular, we describe method invocation directly by substitution. We believe that the semantics of objects given below is natural and suggestive of common implementation techniques.

Primitive semantics

Let $o \equiv [l_i = \varsigma(x_i)b_i^{\ i\in 1..n}]$	(l_i distinct)			
o	is an object with method names l_i and methods $\varsigma(x_i)b_i$			
$o.l_j$	$\rightsquigarrow$	$b_j\{x_j \leftarrow o\}$	$(j\in 1..n)$	selection / invocation
$o.l_j \Leftarrow \varsigma(y)b$	$\rightsquigarrow$	$[l_j = \varsigma(y)b,\ l_i = \varsigma(x_i)b_i^{\ i\in(1..n)-\{j\}}]$	$(j\in 1..n)$	update / override

Notation We use indexed notation of the form $\Phi_i^{\ i\in 1..n}$ to denote sequences $\Phi_1,...,\Phi_n$. The symbol "$\rightsquigarrow$" means "rewrites to". We write $b\{x\}$ to highlight that x may occur free in b. The substitution of a term c for the free occurrences of x in b is written $b\{x \leftarrow c\}$, or $b\{c\}$ when x is clear from context. We use "$\triangleq$" for "equal by definition", "$\equiv$" for "syntactically identical", and "=" for "provably equal" when applied to two terms.

An object is a collection of components $l_i = a_i$, for distinct labels l_i and associated methods a_i; the order of these components does not matter. The object containing a given method is called the method's *host* object. The letter ς (sigma) is used as a binder for the self parameter of a method; $\varsigma(x)b$ is a method with self parameter x, to be bound to the host object, and body b.

A method invocation is written $o.l_j$, where l_j is a label of o. It reduces to the result of the substitution of the host object for the self parameter in the body of the method named l_j.

A method override is written $o.l_j \Leftarrow \varsigma(y)b$. The intent is to replace the method named l_j of o with $\varsigma(y)b$; this is a single operation that involves a construction binding y in b. A method override reduces to a copy of the host object where the overridden method has been replaced by the overriding one. The semantics of override is functional; an override on an object produces a modified copy of the object. In order to make the formal treatment easier we avoid investigating an imperative operational semantics. However, the type theory we develop later is sound, we believe, even for an imperative interpretation of field update and method override.

Self-substitution is at the core of the primitive semantics. Because of this, it is easy to define non-terminating computations without explicit use of recursion:

let $o \triangleq [l = \varsigma(x)x.l]$ then $o.l \rightsquigarrow x.l\{x \leftarrow o\} \equiv o.l \rightsquigarrow ...$

More interestingly, it is possible for a method to return or modify self:

let $o' \triangleq [l = \varsigma(x)x]$ then $o'.l \rightsquigarrow x\{x \leftarrow o'\} \equiv o'$

let $o'' \triangleq [l = \varsigma(y)\ (y.l \Leftarrow \varsigma(x)x)]$ then $o''.l \rightsquigarrow (o''.l \Leftarrow \varsigma(x)x) \rightsquigarrow o'$

We place particular emphasis on the ability to modify self, as illustrated by this last example. In object-oriented languages, it is very common for a method to modify components of self, although these components are normally value fields and not other methods. Generalizing, we allow methods to override other methods of self, or even themselves. This feature does not significantly complicate the problems that we address. Method override is exploited in rather interesting examples that seem difficult to emulate in other calculi.

We should stress that our choice of primitives is not without alternatives.

For example, we could have tried to provide operations to add and remove methods from objects, from which override could be defined [Cardelli, Mitchell 1991]. However, we feel that override is an important operation that it is characterized by peculiar typing rules; we prefer to study it directly and not to explain it away at an early stage.

Similarly, the choice of objects with a fixed number of components, instead of extensible ones [Wand 1987; Mitchell, Honsell, Fisher 1993; Abadi 1994], is a conscious one. Without ruling out future work on extensible objects, we feel that fixed-size objects are easier to handle, particularly in the later stages of our type-theoretical development.

Finally, we do not provide an operation to extract a method from an object as a function. As we shall see, such an operation is incompatible with object subsumption in typed calculi. Methods are inseparable from objects and cannot be recovered as functions; this consideration inspired the use of a specialized ς-notation instead of the familiar λ-notation for parameters.

1.2 Derived Semantics

An inspection of the primitive semantics reveals close similarities between objects and records of functions. It is natural then to try to define objects in terms of records and functions. The correct definition is, however, not evident; we describe a few options.

All implementations of standard (single-dispatch) object-oriented languages are based on self-application so we examine this option first. In the self-application semantics [Kamin 1988], methods are functions, objects are records, object selection is record selection plus self-application, and update is simply update. Records themselves can be encoded as functions over a domain of labels. We write $\langle l_i=a_i \ ^{i\in 1..n}\rangle$ for the record with labels l_i and fields a_i; we write $r\cdot l_j$ for record selection (extracting the l_j component of r), and $r\cdot l_j:=b$ for functional update (producing a copy of r with the l_j component replaced by b).

Self-application semantics

For $o \equiv [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}]$		(l_i distinct)			
o	$\triangleq$	$\langle l_i=\lambda(x_i)b_i \ ^{i\in 1..n}\rangle$			
$o.l_j$	$\triangleq$	$o\cdot l_j(o)$	=	$b_j\{x_j\leftarrow o\}$	$(j\in 1..n)$
$o.l_j \Leftarrow \varsigma(y)b$	$\triangleq$	$o\cdot l_j:=\lambda(y)b$	=	$[l_j=\varsigma(y)b, l_i=\varsigma(x_i)b_i \ ^{i\in(1..n)-\{j\}}]$	$(j\in 1..n)$

The simple equalities shown above, based on the standard β-reduction of λ-terms, reveal that the self-application semantics matches the primitive semantics. Hence, untyped objects can be faithfully interpreted using λ-abstraction, application, and record constructions. Records can be further reduced to pure λ-terms. If we are concerned only with untyped object calculi, little else needs to be said.

Unfortunately, the match between primitive and self-application semantics does not directly extend to typed calculi. By interpreting ς-binders directly as λ-binders, the self-application semantics causes the type of each method to be contravariant in the host object type. The contravariance then blocks expected subtyping relations. For example, the object type Point with x,y integer fields is interpreted as the recursive record type Point = ⟨x,y: Point→Int⟩, where we write $\langle l_i:B_i \ ^{i\in 1..n}\rangle$ for the type of records

with labels l_i and types B_i ($i \in 1..n$). But the type Point = $\langle x,y: \text{Point} \to \text{Int} \rangle$ does not include as a subtype ColorPoint = $\langle x,y: \text{ColorPoint} \to \text{Int}, c: \text{ColorPoint} \to \text{Color} \rangle$, which is the interpretation of the type of points with color.

In view of these difficulties with typing, alternative encodings of objects have been investigated. One immediate idea is to try to "hide" the self parameters so that their types do not appear in contravariant position. This can be achieved by the use of recursive definitions, binding all the self parameters recursively to the object itself [Cardelli 1988]. Note that ς is not interpreted as λ in this semantics:

Recursive-record semantics

For $o \equiv [l_i = \varsigma(x_i) b_i^{\ i \in 1..n}]$	(l_i distinct)			
o	$\triangleq$	$\mu(x)\langle l_i = b_i\{x_i \leftarrow x\}^{\ i \in 1..n}\rangle$	$= \langle l_i = b_i\{x_i \leftarrow o\}^{\ i \in 1..n}\rangle$	
$o.l_j$	$\triangleq$	$o \cdot l_j$	$= b_j\{x_j \leftarrow o\}$	$(j \in 1..n)$

Here, the semantics of method invocation matches the primitive semantics. Moreover, the expected subtyping relations are validated, because the troublesome contravariant parameters are "hidden". If we require only an object calculus without override, then recursive-record semantics will do. Unfortunately, override does not work as expected in this semantics: the most plausible definition appears to be the following:

$o.l_j \Leftarrow \varsigma(y)b$	$\triangleq$	$\mu(x)\, o \cdot l_j := (\lambda(y)b)(x)$	$(j \in 1..n)$
	$=$	$\langle l_j = b\{y \leftarrow (o.l_j \Leftarrow \varsigma(y)b)\},\ l_i = b_i\{x_i \leftarrow o\}^{\ i \in (1..n)-\{j\}}\rangle$	
	$\not\equiv$	$\mu(x)\langle l_j = b\{y \leftarrow x\}, l_j = b_j\{x_j \leftarrow x\}^{\ i \in (1..n)-\{j\}}\ \rangle$	

The result of overriding a method is no longer in the form of an object: record update fails to update "inside the μ" so that all the other methods can become aware of the update. Given the above semantics we have, for example:

$o \triangleq [l_1 = \varsigma(x)3,\ l_2 = \varsigma(x)x.l_1]$

$p \triangleq o.l_1 \Leftarrow \varsigma(x)5$

with primitive semantics	$p.l_1 = 5$	$p.l_2 = 5$
with recursive-record semantics	$p.l_1 = 5$	$p.l_2 = 3$

Therefore, the recursive-record semantics is not adequate. We can argue that the fixpoint operator has been used too soon, and that override has no chance of working once recursion is frozen. In contrast, recursion in the self-application semantics is open-ended: self-application occurs only at method invocation time, not at object-construction time.

As an attempt to salvage the recursive-record semantics, and its nice typing properties, we can try to delay the application of fixpoints. We can replace the μ in that semantics of objects with a λ. Objects are then "record generators" that can be converted to recursive records, when needed, by applying a fixpoint operator ($\mathbf{Y}$). In this way, we obtain the generator semantics of objects [Cook 1989a]:

Generator semantics

For $o \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)						
o	$\triangleq$	$\lambda(x)\langle l_i=b_i\{x_i\leftarrow x\}\ ^{i\in 1..n}\rangle$				
$o.l_j$	$\triangleq$	$\mathbf{Y}(o)\cdot l_j$	$=$	$b_j\{x_j\leftarrow\mathbf{Y}(o)\}$	$\not\equiv$	$b_j\{x_j\leftarrow o\}$ $(j\in 1..n)$
$o.l_j\Leftarrow\varsigma(y)b$	$\triangleq$	$\lambda(x)\ o\cdot l_j:=(\lambda(y)b)(x)$	$=$	$[l_j=\varsigma(y)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$		$(j\in 1..n)$

This semantics works for update, but does not match the primitive semantics for method invocation. Put in the best light, it distinguishes between records and record generators, to the effect that method execution can be performed only after the application of **Y**, and method override only before. For example, after a method invocation returning self, it is no longer possible to perform an update. As a special case, an object cannot update itself through its own methods. Since self-update on fields is a common and important operation, one has to introduce a distinction between field update and method update in order to allow at least the former. Many such refinements of the generator semantics are possible.

Class semantics is a combination of all of the above techniques. It separates the fields from the proper methods of an object. Collectively, the fields represent the state of the object. The methods take the state as a parameter (as in self-application semantics), and are bound to each other by recursion with no possibility of override (as in recursive-record semantics). Objects and object generators are distinguished (as in generator semantics); here we describe only the object part.

Class semantics

For $o \equiv [l_i=\varsigma(x)b_i\ ^{i\in 1..n},\ l_j=\varsigma(x)b_j\ ^{j\in n+1..p}]$ (l_i,l_j distinct; the b_i are fields, that is, do not use x)					
o	$\triangleq$	$\langle s=\langle l_i=b_i\ ^{i\in 1..n}\rangle,\ m=\mu(m)\langle l_i=\lambda(s)b_j'\ ^{i\in 1..n}\rangle\rangle$			(for appropriate b_j')
$o.l_i$	$\triangleq$	$o\cdot s\cdot l_i$	$=$	b_i	$(i\in 1..n)$
$o.l_j$	$\triangleq$	$o\cdot m\cdot l_j(o\cdot s)$	$=$	$b_j'\{s\leftarrow o\cdot s,\ m\leftarrow o\cdot m\}$	$(j\in n+1..p)$
$o.l_i:=b$	$\triangleq$	$\langle s=(o\cdot s\cdot l_i:=b),\ m=o\cdot m\rangle$			$(i\in 1..n)$

The separation of state from methods causes self to split into two parts. Each method body b_j must be transformed into an appropriate new body b_j', as follows: for state extraction through self, $x.l_i$ becomes $s.l_i$; for method invocation through self, $x.l_j$ becomes $m.l_j$; for other uses of self, x must be repackaged into $\langle s=s, m=m\rangle$. These transformations are hard to perform mechanically, in general, but Hofmann and Pierce have studied type-driven techniques [Hofmann, Pierce 1994]. If these transformations are performed correctly, class semantics implements the primitive semantics, except for method override.

One advantage of this rather complex encoding is that it corresponds well to the behavior of class-based languages. Moreover, it can be extended to a typed encoding [Pierce, Turner 1994]. Finally, class semantics becomes considerably simpler with imperative update, since the splitting and repackaging of self is no longer necessary.

1.3 Denotational Semantics

Denotational semantics provides another alternative for explaining objects in terms of records. The denotational semantics we give in section 9 can be understood as encoding typed objects as records that are typable in an extended type system. Intuitively, it is based on the self-application semantics. An object is seen as a record of functions, and an object type as a certain record type.

Let us write $[l_i:B_i\ ^{i\in 1..n}]$ for the type of objects with labels l_i and methods of result type B_i ($i\in 1..n$). In the naive self-application semantics, the object type Point $\triangleq$ [x,y: Int] is interpreted as the record type obtained by solving the type equation Point = ⟨x,y: Point→Int⟩. But this type does not include the solution of ColorPoint = ⟨x,y: ColorPoint→Int, c:ColorPoint→Color⟩, which is the interpretation of ColorPoint $\triangleq$ [x,y: Int, c: Color].

Semantically, we remedy this flaw by resorting to the rich vocabulary of type constructions available in models. Specifically, we interpret the type Point as the union of all the solutions to the equations of the form X = ⟨x,y: X→Int, ... ⟩, including for example X = ⟨x,y: X→Int⟩ and X = ⟨x,y: X→Int, c:X→Color⟩. With this definition, ColorPoint is forced to be a subtype of Point. The denotational semantics is based on the simple idea just described; the details necessary are, as usual, intricate. The semantics is based on a metric approach, applied to complete uniform partial equivalence relations over a Scott domain. It provides a justification for our type theories and our equational theories, and to some extent guided us in their choice.

Returning to the world of syntax, we can reformulate the denotational semantics within a typed calculus. An existential quantifier over all possible extensions of a record type replaces the semantic union operator. With this translation ColorPoint is not a subtype of Point, but there is a canonical coercion from ColorPoint to Point. (See section 10.)

In summary, we have found a faithful interpretation of the primitive semantics of untyped objects within the pure untyped λ-calculus (and any of its models), using the self-application semantics. However, we were not able to find a faithful syntactic intrepretation of the primitive semantics that would model object types adequately while preserving object subtyping. Therefore, we develop a denotational account of object types in terms of record types. This denotational account suggests a syntactic interpretation that does not preserve object subtyping.

In the core of this paper we take objects as a primitive notion, and we give them direct typing rules, equational theories, and denotational semantics. However, we introduce a minimal number of primitive notions, and strive to find the simplest object calculus from which more complex object calculi can be syntactically derived.

1.4 Paper Outline

It is convenient to start within an untyped framework, to assess quickly the potentials of object calculi with selection and override. In section 2 we give a deeper look at the untyped calculus sketched in section 1.1 and we discover that we are able to encode the λ-calculus in it. We show four interesting untyped examples that have guided our investigation of type systems.

In section 3 we study first-order type systems and an equational theory of objects. We find that the simply-typed object calculus can encode the simply-typed λ-calculus with fixpoints, and that the simply-typed object calculus with recursive types can encode the untyped λ-calculus.

In section 4 we introduce subsumption, in the form of object subtyping, and a corresponding equational theory. We also discuss some typing rules that are unsound in presence of subsumption.

In section 5 we examine recursion and we provide preliminary typings for the untyped examples. We conclude by noting that recursively defined object types do not allow some desirable subsumptions. To solve this problem we move up to second-order type systems.

In section 6 we review universal and existential type quantifiers, and we introduce a (definable) *Self* quantifier. We obtain a second-order calculus that includes the first-order object types of section 3. The object types cannot be reduced to more primitive concepts, but more elaborate object constructs, studied in sections 7 and 8, can be obtained from them. In section 7, the combination of the Self quantifier with object types gives us the subsumptions we were looking for, and allows us to typecheck the examples from section 2. In section 8, we consider the delicate issue of overriding methods that return self.

Section 9 provides a semantics for the calculus of section 6, giving direct meaning to object types, and interpreting the remaining first-order and second-order constructs in a standard way. Section 10 defines a translation of objects and object types into a calculus without subtyping; in this translation, subsumption is interpreted by explicit coercions.

An appendix summarizes the relevant primitive and derived rules of our main calculi.

2. Untyped Calculi

In this section we investigate the expressiveness of the untyped calculus. We assume the primitive semantics as the intended semantics of objects.

2.1 The Untyped ς-calculus

The following formal syntax describes a pure object calculus without functions. In later sections, this calculus is the basis for our typed calculi.

Syntax of the ς-calculus

a,b ::=			terms
	x		variable
	$[l_i=\varsigma(x_i)b_i^{\ i\in 1..n}]$	(l_i distinct)	object
	a.l		field selection / method invocation
	$a.l \Leftarrow \varsigma(x)b$		field update / method override

Here, an object $[l_i=\varsigma(x_i)b_i^{\ i\in 1..n}]$ has method names l_i and methods $\varsigma(x_i)b_i$. In a method ς(x)b, x is the self variable and b is the body.

Notation

- $o.l_j:=b$ stands for $o.l_j \Leftarrow \varsigma(y)b$, for an unused y. We call $o.l_j:=b$ an *update* operation.
- [..., l=b, ...] stands for [..., l=ς(y)b, ...], for an unused y. We call l=b a *field*.
- We identify ς(x)b with $\varsigma(y)(b\{x\leftarrow y\})$, for any y not occurring free in b.

We repeat the primitive semantics of objects below, for convenience. As is customary, we do not make error conditions explicit. We simply assume that objects and methods are used consistently, and that otherwise computations stop without producing a result (so, for example, [].l does not rewrite to anything).

Reduction rules for objects

Let $o \equiv [l_i=\varsigma(x_i)b_i^{\ i\in 1..n}]$ (l_i distinct)

$o.l_j$	$\rightsquigarrow$	$b_j\{x_j\leftarrow o\}$	$(j\in 1..n)$
$o.l_j \Leftarrow \varsigma(y)b$	$\rightsquigarrow$	$[l_j=\varsigma(y)b, l_i=\varsigma(x_i)b_i^{\ i\in(1..n)-\{j\}}]$	$(j\in 1..n)$

We can derive an untyped equational theory from the untyped reduction rules. We add rules for symmetry, transitivity, and congruence; we reformulate the reduction rules as equalities.

Equational theory

(Eq Symm)
$$\frac{b \leftrightarrow a}{a \leftrightarrow b}$$

(Eq Trans)
$$\frac{a \leftrightarrow b \quad b \leftrightarrow c}{a \leftrightarrow c}$$

(Eq x)
$$\frac{}{x \leftrightarrow x}$$

(Eq Object)
$$\frac{b_i \leftrightarrow b_i' \quad \forall i \in 1..n}{[l_i = \varsigma(x_i) b_i^{\,i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i) b_i'^{\,i \in 1..n}]} \quad (l_i \text{ distinct})$$

(Eq Select)
$$\frac{a \leftrightarrow a'}{a.l \leftrightarrow a'.l}$$

(Eq Override)
$$\frac{a \leftrightarrow a' \quad b \leftrightarrow b'}{a.l \Leftarrow \varsigma(x)b \leftrightarrow a'.l \Leftarrow \varsigma(x)b'}$$

(Eval Select) (where $a \equiv [l_i = \varsigma(x_i) b_i^{\,i \in 1..n}]$)
$$\frac{j \in 1..n}{a.l_j \leftrightarrow b_j\{x_j \leftarrow a\}}$$

(Eval Override) (where $a \equiv [l_i = \varsigma(x_i) b_i^{\,i \in 1..n}]$)
$$\frac{j \in 1..n}{a.l_j \Leftarrow \varsigma(x)b \leftrightarrow [l_j = \varsigma(x)b, l_i = \varsigma(x_i) b_i^{\,i \in (1..n)-\{j\}}]}$$

To complete the formal syntax of the ς-calculus we give the definitions of free variables (FV) and substitution ($b\{x \leftarrow a\}$) for ς-terms.

Object scoping and substitution

$FV(\varsigma(y)b)$	$\triangleq$	$FV(b)-\{y\}$
$FV(x)$	$\triangleq$	$\{x\}$
$FV([l_i = \varsigma(x_i) b_i^{\,i \in 1..n}])$	$\triangleq$	$\bigcup^{i \in 1..n} FV(\varsigma(x_i)b_i)$
$FV(o.l)$	$\triangleq$	$FV(o)$
$FV(o.l \Leftarrow \varsigma(y)b)$	$\triangleq$	$FV(o) \cup FV(\varsigma(y)b)$

$(\varsigma(y)b)\{x \leftarrow a\}$	$\triangleq$	$\varsigma(y')(b\{y \leftarrow y'\}\{x \leftarrow a\})$	for $y' \notin FV(\varsigma(y)b) \cup FV(a) \cup \{x\}$
$x\{x \leftarrow a\}$	$\triangleq$	a	
$y\{x \leftarrow a\}$	$\triangleq$	y	for $y \neq x$
$[l_i = \varsigma(x_i) b_i^{\,i \in 1..n}]\{x \leftarrow a\}$	$\triangleq$	$[l_i = (\varsigma(x_i) b_i)\{x \leftarrow a\}^{\,i \in 1..n}]$	
$(o.l)\{x \leftarrow a\}$	$\triangleq$	$(o\{x \leftarrow a\}).l$	
$(o.l \Leftarrow \varsigma(y)b)\{x \leftarrow a\}$	$\triangleq$	$(o\{x \leftarrow a\}).l \Leftarrow ((\varsigma(y)b)\{x \leftarrow a\})$	

The connection between equality ($\leftrightarrow$) and reduction ($\rightsquigarrow$) is clarified by the following theorem:

Theorem 2.1-1 (Church-Rosser)

We say that a one-step-reduces to b, and write $a \to b$, if a is c[a'], b is c[b'], and $a' \rightsquigarrow b'$, where c is any context. Let $\twoheadrightarrow$ be the reflexive transitive closure of $\to$. The relation $\twoheadrightarrow$ is Church-Rosser, and if $a \leftrightarrow b$ then there exists c such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.

□

The proof of this result follows the method of Tait and Martin-Löf [Barendregt 1985]. The sequence of definitions and lemmas is standard.

2.2 Functions as Objects

It would be possible to add ordinary λ-terms to our object calculus. However, having two similar variables binders (λ and ς) in a small calculus seems excessive. We could replace ς with λ, identifying methods with functions. We feel, instead, that the ς-binders have a special status. First, as we have seen, they can be given a simple and direct rewrite semantics, instead of an indirect semantics involving both λ-abstraction and application. Second, ς-binders by themselves have a surprising expressive power; we show below that they can be as expressive as λ-binders.

We define a translation ⟪ ⟫ from λ-terms to pure objects:

Translation of the untyped λ-calculus

$$\ll x \gg \triangleq x$$

$$\ll b(a) \gg \triangleq \ll b \gg \bullet \ll a \gg \qquad \text{where } p \bullet q \triangleq (p.arg := q).val$$

$$\ll \lambda(x)b\{x\} \gg \triangleq [arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\, \ll b\{x\} \gg \{x \leftarrow x.arg\}]$$

The idea is that an application ⟪b(a)⟫ first stores the argument ⟪a⟫ into a known place (the field arg, whose initial value is unimportant) inside of ⟪b⟫, and then invokes a method of ⟪b⟫ that can access the argument through self. For example:

$$\ll (\lambda(x)x)(y) \gg \equiv ([arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\, x.arg].arg := y).val = y$$

Note that the translation maps nested λ's to nested ς's: although every method has a single self parameter, we can emulate functions with multiple parameters.

Renaming (α-conversion) of λ-bound variables is valid under the translation (up to α-conversion) because of the renaming properties of ς-binders. We can verify that β-conversion is valid under the translation as well:

$$let\ o \equiv [arg = \ll a \gg,\ val = \varsigma(x)\, \ll b\{x\} \gg \{x \leftarrow x.arg\}]$$

$$\begin{aligned} &\ll (\lambda(x)b\{x\})(a) \gg \\ &\quad \equiv ([arg = \varsigma(x)\, x.arg,\ val = \varsigma(x)\, \ll b\{x\} \gg \{x \leftarrow x.arg\}].arg := \ll a \gg).val \\ &\quad = o.val = (\ll b\{x\} \gg \{x \leftarrow x.arg\})\{x \leftarrow o\} = \ll b\{x\} \gg \{x \leftarrow o.arg\} = \ll b\{x\} \gg \{x \leftarrow \ll a \gg\} \\ &\quad = \ll b\{a\} \gg \end{aligned}$$

However, η-conversion is not valid under the translation, since not every object has the form [arg = ..., val = ...].

This translation can be extended to provide a natural interpretation of λ-terms with default parameters and with call-by-keyword. We write λ(x=c)b{x} for a function with a single parameter x with default c. We write f(a) for a normal application of f to a, and f() for an application of f to its default. For example, (λ(x=c)x)() = c and (λ(x=c)x)(a) = a. The interpretation of λ-terms with a single default parameter is:

$\langle\!\langle \lambda(x=c)b\{x\} \rangle\!\rangle \triangleq [arg = \langle\!\langle c \rangle\!\rangle, val = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.arg\}]$
$\langle\!\langle b(a) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle \bullet \langle\!\langle a \rangle\!\rangle$
$\langle\!\langle b() \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle .val$

We write $\lambda(x_i=c_i\ ^{i\in 1..n})b$ for a function with multiple parameters x_i with defaults c_i. The application $f(y_i=a_i\ ^{i\in 1..m})$ can provide fewer parameters than f expects, and in any order. The association of the provided actuals to the formals is made by the names x_i, with the defaults being used for the missing actuals. The interpretation of λ-terms with call-by-keyword and default parameters is:

$\langle\!\langle \lambda(x_i=c_i\ ^{i\in 1..n})b\{x_i\ ^{i\in 1..n}\} \rangle\!\rangle \triangleq$	$x_i \neq val,\ z \notin FV(b)$
$[x_i = \langle\!\langle c_i \rangle\!\rangle^{i\in 1..n}, val = \varsigma(z)\, \langle\!\langle b\{x_i\ ^{i\in 1..n}\} \rangle\!\rangle \{x_i \leftarrow z.x_i\ ^{i\in 1..n}\}]$	
$\langle\!\langle b(y_i=a_i\ ^{i\in 1..m}) \rangle\!\rangle \triangleq (\langle\!\langle b \rangle\!\rangle .y_i := \langle\!\langle a_i \rangle\!\rangle^{i\in 1..m}).val$	$y_i \neq val$

2.3 Fixpoints

As a consequence of the translation of λ-terms into pure objects, we obtain object-oriented versions of all the encodings that are possible within the λ-calculus. None of these encodings seem, however, particularly inspiring; more direct object-oriented encodings can usually be found. Such is the case, for example, for fixpoint operators. The encoding given below is much simpler than others that can be obtained by translation of λ-terms:

A fixpoint operator

fix ≜ [arg = ς(x) x.arg, val = ς(x) ((x.arg).arg := x.val).val]

We can verify the fixpoint property as follows, recalling that $p \bullet q \equiv (p.arg := q).val$ is the encoding of function application:

$$
\begin{aligned}
fix_f &\triangleq fix.arg := f \\
&= [arg = f,\ val = \varsigma(x)\ ((x.arg).arg := x.val).val] \\
\\
fix \bullet f &\equiv fix_f.val \\
&= ((fix_f.arg).arg := fix_f.val).val \\
&= (f.arg := fix \bullet f).val \\
&\equiv f \bullet (fix \bullet f)
\end{aligned}
$$

In particular, if we add a constant *fix* to the λ-calculus, and we set $\langle\!\langle fix \rangle\!\rangle \triangleq$ fix, then $\langle\!\langle fix(f) \rangle\!\rangle = \langle\!\langle f(fix(f)) \rangle\!\rangle$.

As a curiosity, observe the resemblance of fix with:

$$\langle\!\langle \lambda(x)x(x) \rangle\!\rangle \equiv$$
$$[arg = \varsigma(x)\ x.arg,$$
$$\ val = \varsigma(x)\ ((x.arg).arg := x.arg).val]$$

Furthermore, we can provide a translation for $\mu(x)b\{x\}$ that is more compact than its natural definition as $\langle\!\langle fix(\lambda(x)b\{x\}) \rangle\!\rangle$:

$$\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle \triangleq$$
$$[rec = \varsigma(x)\ \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.rec\}].rec$$

with the unfolding property:

$$[rec = \varsigma(x)\ \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.rec\}].rec$$
$$= \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.rec\}\{x \leftarrow [rec = \varsigma(x)\ \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.rec\}]\}$$
$$= \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow [rec = \varsigma(x)\ \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.rec\}].rec\}$$
$$= \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow \langle\!\langle \mu(x)b\{x\} \rangle\!\rangle\}.$$

The recursion operators described above use only object primitives. With λ-abstraction and application, we can write the fixpoint operator of [Mitchell, Honsell, Fisher 1993]:

$$fix' \triangleq \lambda(f)\ [rec = \varsigma(s)\ f(s.rec)].rec$$

whose translation is similar, but not identical, to our fix:

$$\langle\!\langle fix' \rangle\!\rangle \equiv [arg = \varsigma(x)x.arg,\ \ val = \varsigma(x)\ [rec = \varsigma(s)\ ((x.arg).arg:=s.rec).val].rec\ ]$$

2.4 Four Examples

The general problem we are confronting is to find useful type systems for the untyped ς-calculus. We now examine some examples that can be easily written in the untyped calculus, but pose interesting typing difficulties. Our examples involve updating self, so neither the recursive-record semantics nor the generator semantics would be adequate. One of the examples is based on overriding a proper method, so the class semantics does not apply. The typing requirements range from very basic ones, typing geometric points, to very sophisticated ones, typing an object-oriented version of the numerals inspired by Scott numerals [Wadsworth 1980]. For the impatient, typed versions of these examples can be found in sections 5.3, 7.2, and 8.3.2.

In these examples, we use boolean, integer, and real constants; later it should become clear how to embed or encode basic data types within our calculi. The notation of the untyped λ-calculus is also used freely, since it can be encoded.

2.4.1 Movable Points

A classic object-oriented example involves geometric points with a move method that updates the point coordinates.

Movable points

$$\begin{aligned}&\text{origin}_1 \triangleq\\ &\quad [x = 0,\\ &\quad\ \text{mv_x} = \varsigma(s)\ \lambda(dx)\ s.x{:=}s.x+dx]\\ &\text{origin}_2 \triangleq\\ &\quad [x = 0,\ y = 0,\\ &\quad\ \text{mv_x} = \varsigma(s)\ \lambda(dx)\ s.x{:=}s.x+dx,\\ &\quad\ \text{mv_y} = \varsigma(s)\ \lambda(dy)\ s.y{:=}s.y+dy]\end{aligned}$$

For example, we can define $\text{unit}_2 \triangleq \text{origin}_2.\text{mv_x}(1).\text{mv_y}(1)$, and compute $\text{unit}_2.x = 1$.

We would like to obtain a type system where a movable two-dimensional point, such as origin_2, can be accepted in any context expecting a movable one-dimensional point, such as origin_1. One potential difficulty concerns the contravariance of the types of methods in the type of self, as discussed in section 1.2.

2.4.2 Backup Methods

We now give a simple example that illustrates two techniques: storing self and using two different versions of self at once. We define an object that is able to keep backup copies of itself, for example as an auditing trail. This object has a backup method, and a retrieve method that returns the last backup:

Objects with backup

$$\begin{aligned}o \triangleq [&\text{retrieve} = \varsigma(s_1)\ s_1\ ,\\ &\text{backup} = \varsigma(s_2)\ s_2.\text{retrieve} \Leftarrow \varsigma(s_1)\ s_2\ ,\\ &\textit{(additional fields and methods)}\ ]\end{aligned}$$

The initial retrieve method is set to return the initial object. Whenever the backup method is invoked, it stores a copy of self into retrieve. Note that backup stores the self s_2 that is current at backup-invocation time, not the self s_1 that will be current at retrieve-invocation time. For example:

$$o' \triangleq o.\text{backup} = [\text{retrieve} = \varsigma(s_1)\ o, ...]$$

Later, possibly after modifying the additional fields and methods, we can extract the object that was most recently backed up. The retrieve method returns the self that was current at the last invocation of backup, as desired.

$$o'.\text{retrieve} = o$$

We can cascade invocations of the retrieve method to recover older and older backups, eventually converging to the initial object.

2.4.3 Object-Oriented Natural Numbers

The technique of storing self, illustrated in the previous example, comes up in another interesting situation. We would like to define the object-oriented natural numbers, that is, objects that respond to the methods iszero (test for zero), pred (predecessor), and succ (successor), and behave like the natural numbers.

We need to define only the numeral zero, since its succ method will generate all the other numerals. Obviously, the numeral zero should answer true to the iszero question, and zero to the pred question.

The succ method is not so easy to manufacture. When succ is invoked on zero, it should modify zero so that it becomes one. That is, succ should modify self so that it answers false to the iszero question, and zero to the pred question. Moreover, succ should be such that when invoked again on numeral one, it produces an appropriate numeral two, etc. Hence, for any numeral, succ should update iszero to answer false, and should update pred to return the self that is current when succ is invoked.

```
zero ≜
    [iszero = true,
     pred = ς(x) x
     succ = ς(x) (x.iszero := false).pred := x ]
```

Here the body of succ consists of two cascaded updates to self. We can verify, with a few tests, that the operational semantics of natural numbers is well represented.

Instead of defining numbers with iszero, pred, and succ, we can define numbers with only two methods: succ and case. This gives a simpler, although more opaque, encoding of the natural numbers that does not depend on booleans:

Object-oriented numerals

```
zero ≜
    [case = λ(z) λ(s) z,
     succ = ς(x) x.case := λ(z) λ(s) s(x) ]
```

The case method is in fact a field containing a function of two arguments. For a numeral n, the first argument is returned if n is zero, otherwise the second argument is applied to the predecessor of n; that is, n.case(a)(f) equals a if n is zero, and equals f(x) if n is nonzero and x is its predecessor. The predecessor of n is obtained by the now familiar technique of capturing a previous self. We can compute:

```
one  ≜  zero.succ  =  [case = λ(z) λ(s) s(zero),  succ = (unchanged)]
two  ≜  one.succ   =  [case = λ(z) λ(s) s(one),   succ = (unchanged)]
```

Moreover, we can recover the iszero and pred methods as functions:

```
iszero  ≜  λ(n) n.case(true)(λ(p)false)
pred    ≜  λ(n) n.case(zero)(λ(p)p)
```

In this example, as in the one in the previous section, current self and future self can be statically nested and handled at once. Because of this, it is critical that self be a named parameter. Providing a single keyword “self” in the scope of a method, as done in many object-oriented languages, would not be sufficient.

2.4.4 A Calculator

Our final example is that of a calculator object. We exploit the ability to override methods to record the pending arithmetic operation. When an operation add or sub is entered, the equals method is overridden with code for addition or subtraction. The first two components (arg, acc) are needed for the internal operation of the calculator, while the other four (enter, add, sub, equals) provide the user interface.

Calculator

```
calculator ≜
    [arg = 0.0,
     acc = 0.0,
     enter = ς(s) λ(n) s.arg := n,
     add = ς(s) (s.acc := s.equals).equals ⇐ ς(s') s'.acc+s'.arg,
     sub = ς(s) (s.acc := s.equals).equals ⇐ ς(s') s'.acc-s'.arg,
     equals = ς(s) s.arg
    ]
```

A reset operation could be added to clear the state and restore the initial equals method.

This definition is slightly subtle; it is meant to provide the following calculator-style behavior:

calculator .enter(5.0) .equals	=	5.0
calculator .enter(5.0) .sub .enter(3.5) .equals	=	1.5
calculator .enter(5.0) .add .add .equals	=	15.0

3. First-Order Calculi

We now begin to investigate the type theory of the ς-calculus. We start with a simple first-order type system with object types.

We compose our typed systems from *formal system fragments*. Each fragment is named Δ_s for an appropriate subscript *s*. These fragments can be reassembled to form standard typed calculi, some of which are listed in the appendix. Each fragment consists of a set of related rules. Each rule has a number of antecedent judgments above a horizontal line and a single conclusion judgment below the line. Each judgment has the form $E \vdash \mathfrak{I}$, for a typing environment E and an assertion $\mathfrak{I}$ depending on the judgment. An antecedent of the form "$E,E_i \vdash \mathfrak{I}_i \quad \forall i \in 1..n$" is an abbreviation for n antecedents "$E,E_1 \vdash \mathfrak{I}_1$ ... $E,E_n \vdash \mathfrak{I}_n$" if n>0, and if n=0 for "$E \vdash \diamond$", which means that E is well-formed. Instead, a rule containing "$j \in 1..n$" indicates that there are n separate rules, one for each j. Each rule has a name whose first word is determined by the kind of judgment in its conclusion; for example names of the form "(Type ...)" are for rules whose conclusion is a type judgment.

3.1 The Object Fragment

We start with the formal system fragment corresponding to object types. An object of type $[l_i:B_i\ ^{i \in 1..n}]$ can be formed from a collection of n methods whose self parameters have type $[l_i:B_i\ ^{i \in 1..n}]$ and whose bodies have types B_1 ... B_n. We always assume, when writing $[l_i:B_i\ ^{i \in 1..n}]$, that the l_i are distinct and that permutations do not matter. When a method l_i is invoked, it produces a result having the corresponding type B_i. A method can be overridden while preserving the type of its host object.

The ς-bound variables have type annotations equal to the type of their host object. An object type $[l_i:B_i\ ^{i \in 1..n}]$ exhibits only the result types B_i of its methods: it does not explicitly list the types of the ς-bound variables. The types of all these variables are equal to the object type itself, so no information is missing.

Two judgments are used below: type judgments $E \vdash B$ (meaning that B is a well-formed type in the environment E) and value judgments $E \vdash b : B$ (meaning that b has type B in E). Environments contain typing assumptions for variables.

$\Delta_{\mathbf{Ob}}$

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1..n \qquad (l_i \text{ distinct})}{E \vdash [l_i : B_i \ ^{i \in 1..n}]}$$

(Val Object) (where $A \equiv [l_i : B_i \ ^{i \in 1..n}]$)

$$\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i : A) b_i \ ^{i \in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i : B_i \ ^{i \in 1..n}]$)

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x : A) b : A}$$

Notation

- $o.l_j := b$ stands for $o.l_j \Leftarrow \varsigma(y : A) b$, for an appropriate A and some $y \notin FV(b)$.
- [..., l=b, ...] stands for [..., $l = \varsigma(y : A) b$, ...], for an appropriate A and some $y \notin FV(b)$.
- [..., l,m:B, ...] stands for [..., l:B, m:B ...], in examples.
- We identify $\varsigma(x : A) b$ with $\varsigma(y : A)(b\{x \leftarrow y\})$, for any $y \notin FV(b)$.

3.2 A Complete Calculus

We can obtain a well-rounded typed version of the ς-calculus by adding to $\Delta_{\mathbf{Ob}}$ two standard fragments for variables and data algebras. Furthermore, we can add a fragment for typed λ-terms.

The environment judgment $E \vdash \diamond$ is used to construct well-formed environments for variables:

$\Delta_{\mathbf{x}}$

(Env ∅)

$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)

$$\frac{E \vdash A \quad x \notin dom(E)}{E, x : A \vdash \diamond}$$

(Val x)

$$\frac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x : A}$$

Built-in data structures, such as booleans and integers, are given as first-order free algebras. We fix a set Sort of built-in sorts. We also fix families $Op(K_1,...,K_{n+1})$ of operations of arity $(K_1,...,K_n)$ and result K_{n+1}, with $K_i \in Sort$ and $n \geq 0$. We assume that these families are pairwise disjoint.

$\Delta_{\mathbf{K}}$

(Type Const)

$$\frac{E \vdash \diamond \quad K \in Sort}{E \vdash K}$$

(Val Const)

$$\frac{k \in Op(K_{i\ i\in 1..n+1}) \quad E \vdash a_i : K_i \quad \forall i \in 1..n}{E \vdash k(a_{i\ i\in 1..n}) : K_{n+1}}$$

Function types are described in the following fragment:

$\Delta_{\rightarrow}$

(Type Arrow)

$$\frac{E \vdash A \quad E \vdash B}{E \vdash A \rightarrow B}$$

(Val Fun)

$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$$

(Val Appl)

$$\frac{E \vdash b : A \rightarrow B \quad E \vdash a : A}{E \vdash b(a) : B}$$

We now define the calculi:

$\mathbf{Ob_1}$	$\triangleq$	$\Delta_{\mathbf{K}} \cup \Delta_{\mathbf{x}} \cup \Delta_{\mathbf{Ob}}$	the first-order typed ς-calculus with constants
$\mathbf{F_1}$	$\triangleq$	$\Delta_{\mathbf{K}} \cup \Delta_{\mathbf{x}} \cup \Delta_{\rightarrow}$	the first-order typed λ-calculus with constants
$\mathbf{FOb_1}$	$\triangleq$	$\Delta_{\mathbf{K}} \cup \Delta_{\mathbf{x}} \cup \Delta_{\rightarrow} \cup \Delta_{\mathbf{Ob}}$	the first-order typed $\lambda\varsigma$-calculus with constants

3.2.1 Unique Types

The $\mathbf{Ob_1}$ calculus enjoys an important property: every term has a unique type.

Proposition 3.2.1-1 ($\mathbf{Ob_1}$ has unique types)
If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{Ob_1}$, then $A \equiv A'$.
□

The proof is a trivial induction on the derivation of $E \vdash a : A$, and extends to $\mathbf{FOb_1}$.

Unique typing is an obvious property for $\mathbf{Ob_1}$, but small perturbations of the rules do not always preserve it. The property remains true if we omit the type annotation for override, by adopting the rule:

(Val Override') (where $A \equiv [l_i{:}B_{i\ i\in 1..n}]$)

$$\frac{E \vdash a : A \quad E, x{:}A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x)b : A}$$

However, it fails if we omit type annotations for object construction, by adopting the rule:

(Val Object') (where $A \equiv [l_i{:}B_{i\ i\in 1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_{i\ i\in 1..n}] : A}$$

For example, in the modified system, $[l = \varsigma(x)x.l]$ has type $[l{:}A]$ for any A. Still, the convention of omitting ς-binders entirely for methods that do not depend on self is innocuous. For example, $[l=3]$ has unique type $[l{:}Int]$ if Int is the type of 3.

3.3 Equational Theory of Ob_1

We now investigate the equational theory of the Δ_{Ob} fragment, which was implicitly assumed in some of the previous discussion. The equational theories for the other fragments are standard; see the appendix. For simplicity, when assembling a calculus we list only the typing fragments. The corresponding equational fragments are assumed from context since, at least in this paper, they are uniquely determined.

We use a new judgment $E \vdash b \leftrightarrow c : A$ to assert that b and c are equivalent when considered as elements of type A. The first two rules for this new judgment express symmetry and transitivity; reflexivity will be obtained as a derived rule.

$\Delta_{=}$

(Eq Symm)

$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \vdash a \leftrightarrow b : A \qquad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

In examples we assume equational rules for constants, but we do not present these rules formally. There is an obvious rule for variables: a limited form of reflexivity.

$\Delta_{=x}$

(Eq x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x \leftrightarrow x : A}$$

The first three rules for objects are congruence rules, establishing that two expressions are equal when all their corresponding subexpressions are equal. The next two rules are evaluation rules for selection and override, corresponding to the operational semantics of the untyped calculus of section 2.1.

$\Delta_{=Ob}$

(Eq Object) (where $A \equiv [l_i{:}B_i \ ^{i \in 1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i{:}A)b_i \ ^{i \in 1..n}] \ \leftrightarrow \ [l_i = \varsigma(x_i{:}A)b_i' \ ^{i \in 1..n}] : A}$$

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i{:}B_i \ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j \ \leftrightarrow \ a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i{:}B_i \ ^{i \in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \qquad E, x{:}A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b \ \leftrightarrow \ a'.l_j \Leftarrow \varsigma(x{:}A)b' : A}$$

(Eval Select) (where $A \equiv [l_i{:}B_i \ ^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i{:}A')b_i \ ^{i \in 1..n}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \ \leftrightarrow \ b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \ \leftrightarrow\ [l_j = \varsigma(x:A')b,\ l_i = \varsigma(x_i:A')b_i\ ^{i\in(1..n)-\{j\}}] :\ A}$$

For functions, we have the standard theory for the first-order λ-calculus:

$\Delta_{=\rightarrow}$

(Eq Fun)

$$\frac{E,x:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B}$$

(Eq Appl)

$$\frac{E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x:A)b : A \rightarrow B \qquad E \vdash a : A}{E \vdash (\lambda(x:A)b)(a) \ \leftrightarrow\ b\{x \leftarrow a\} : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \rightarrow B \qquad x \notin dom(E)}{E \vdash \lambda(x:A)b(x) \ \leftrightarrow\ b : A \rightarrow B}$$

This simple equational theory is already quite interesting. In record calculi we can safely assume that two records r and r' are equal if they have the same labels $l_i\ ^{i\in 1..n}$ and if the values $r \cdot l_i$ and $r' \cdot l_i$ are equal for all $i \in 1..n$. This property does not hold for objects: two objects may give equal result for all their methods, and still be distinguishable. Consider the following objects:

$$\begin{aligned} A &\triangleq [x:Nat, f:Nat] \\ a:A &\triangleq [x=1, f=\varsigma(s:A)1] \\ b:A &\triangleq [x=1, f=\varsigma(s:A)s.x] \end{aligned}$$

We might think that a and b are equal at type A because $\emptyset \vdash a.x \leftrightarrow b.x : Nat$ and $\emptyset \vdash a.f \leftrightarrow b.f : Nat$. But to prove their equality, the (Eq Object) rule requires showing $\emptyset, s:A \vdash 1 \leftrightarrow s.x : Nat$. This cannot be obtained because we have no assumptions about the value of self, in particular that s.x is currently 1. In fact, self may change and invalidate this assumption about its value. For example, it is possible to distinguish a from b after updating them both with equal values:

$$\begin{aligned} a':A &\triangleq a.x:=2 \\ b':A &\triangleq b.x:=2 \end{aligned}$$

Now we have $\emptyset \vdash a'.f \leftrightarrow 1 : Nat$ while $\emptyset \vdash b'.f \leftrightarrow 2 : Nat$, so asserting $\emptyset \vdash a \leftrightarrow b : A$ would lead to a contradiction.

This example illustrates a fundamental difference between the equational theories of object calculi and record calculi, as well as a fundamental difficulty in reasoning about objects. Still, we would like to say more about object equivalence than $\Delta_{=\mathbf{Ob}}$ allows. For example, we may wish to determine that a and b above are interchangeable in contexts that read or modify only their x components; that is, in contexts where a and b are considered as having type [x:Nat]. This involves an implicitly or explicit assumption that a longer object belongs to a shorter type. We examine this idea in section 4.

3.4 Functions as Objects

The $\mathbf{Ob_1}$ calculus is sufficient to encode $\mathbf{F_1}$, along the lines sketched in section 2.2. Hence, $\mathbf{FOb_1}$ has a built-in redundancy, although a very convenient one. The translation from $\mathbf{F_1}$ to $\mathbf{Ob_1}$ is shown below. Strictly speaking, this translation is defined on type derivations, but for simplicity we write it as a translation of type-annotated λ-terms.

Translation of the first-order λ-calculus

$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$

$\langle\!\langle E,x:A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x:\langle\!\langle A \rangle\!\rangle$

$\langle\!\langle K \rangle\!\rangle \triangleq K$

$\langle\!\langle A \to B \rangle\!\rangle \triangleq [arg:\langle\!\langle A \rangle\!\rangle, val:\langle\!\langle B \rangle\!\rangle]$

$\langle\!\langle x_A \rangle\!\rangle \rho \triangleq \rho(x)$

$\langle\!\langle k_K \rangle\!\rangle \rho \triangleq k$

$\langle\!\langle b_{A\to B}(a_A) \rangle\!\rangle \rho \triangleq$
$(\langle\!\langle b \rangle\!\rangle \rho.arg \Leftarrow \varsigma(x:\langle\!\langle A \to B \rangle\!\rangle)\ \langle\!\langle a \rangle\!\rangle \rho).val$
for $x \notin FV(\langle\!\langle a \rangle\!\rangle \rho)$

$\langle\!\langle \lambda(x:A) b_B \rangle\!\rangle \rho \triangleq$
$[arg = \varsigma(x:\langle\!\langle A \to B \rangle\!\rangle)\ x.arg,$
$val = \varsigma(x:\langle\!\langle A \to B \rangle\!\rangle)\ \langle\!\langle b \rangle\!\rangle \rho\{x \leftarrow x.arg\}]$

$\rho \in Var \to \mathbf{Ob_1}\text{-term}$

$\emptyset(x) \triangleq x$

$(\rho\{y \leftarrow a\})(x) \triangleq$ if x=y then a else $\rho(x)$

It is not difficult to verify that the translation maps valid derivations in $\mathbf{F_1}$ to valid derivations in $\mathbf{Ob_1}$. Typed β-reduction is satisfied by this encoding, but typed η-reduction is not.

In what follows, we describe several extensions of $\mathbf{F_1}$ and $\mathbf{Ob_1}$. Consider a pair of extensions, $\mathbf{F}_*$ and $\mathbf{Ob}_*$. We say that $\mathbf{Ob}_*$ can encode $\mathbf{F}_*$ when there exists a translation mapping derivations of $\mathbf{F}_*$ that do not use the first-order η rule into derivations of $\mathbf{Ob}_*$. (This η rule is (Eval Eta) of section 3.3.) In this sense, $\mathbf{Ob_1}$ can encode $\mathbf{F_1}$.

In addition, $\mathbf{Ob_1}$ can encode an extension of $\mathbf{F_1}$ with a fixpoint operator $fix_A:(A\to A)\to A$ for each A, in such a way that $\langle\!\langle fix_A(f) \rangle\!\rangle = \langle\!\langle f(fix_A(f)) \rangle\!\rangle$. All we need is the fixpoint operator of section 2.3, which is typable within $\mathbf{Ob_1}$.

$\langle\!\langle fix_A \rangle\!\rangle \rho \triangleq$
$[arg = \varsigma(x:\langle\!\langle (A\to A)\to A \rangle\!\rangle)\ x.arg,$
$val = \varsigma(x:\langle\!\langle (A\to A)\to A \rangle\!\rangle)\ ((x.arg).arg := x.val).val]$

Both $\langle\!\langle fix_A \rangle\!\rangle \rho$ and the ς-bound variables of this term have type

$\langle\!\langle (A\to A)\to A \rangle\!\rangle = [arg: [arg: \langle\!\langle A \rangle\!\rangle, val: \langle\!\langle A \rangle\!\rangle], val: \langle\!\langle A \rangle\!\rangle].$

$\mathbf{Ob_1}$, unlike $\mathbf{F_1}$, is not normalizing; this is obvious from the definability of fixpoint operators. In fact, there exist simple divergent terms. For example $[l=\varsigma(x:[l:[]])x.l].l$ is typable as follows:

$\emptyset \vdash \diamond$	by (Env ∅)
$\emptyset \vdash []$	by (Type Object) with n=0
$\emptyset \vdash [l:[]]$	by (Type Object) with n=1
$\emptyset, x:[l:[]] \vdash \diamond$	by (Env x)
$\emptyset, x:[l:[]] \vdash x : [l:[]]$	by (Val x)
$\emptyset, x:[l:[]] \vdash x.l : []$	by (Val Select)

$\emptyset \vdash [l=\varsigma(x:[l:[]])x.l] : [l:[]]$ by (Val Object) with n=1

$\emptyset \vdash [l=\varsigma(x[l:[]])x.l].l : []$ by (Val Select)

4. First-Order Calculi with Subsumption

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods, since the former supports the entire protocol of the latter. Conversely, a context that expects an object with a given method protocol can be filled with an object that has a more extended protocol. We call this notion *subsumption*: an object can subsume another object that has a more limited protocol.

No object calculus can fully justify its existence without some notion of subsumption. This criticism should first be directed to $\mathbf{FOb_1}$, studied in section 3. We should notice that, in accordance with the self-application semantics, there is no difficulty in adding to $\mathbf{FOb_1}$ a new operation that extracts a method from an object and returns it as a function with parameter self. Without a good reason for ruling out this extraction operation, an object calculus would be just an oddly restricted record calculus. As it happens, the idea of extracting a method from an object is uncharacteristic of object-oriented languages and, as we shall see shortly, this is precisely because this operation is fundamentally incompatible with the typing of subsumption.

To address this inadequacy, in this section we define a particular form of subsumption that is induced by a subtyping relation between object types. An object that belongs to a given object type also belongs to any supertype of that type, and can subsume objects in the supertype.

4.1 A Calculus with Subtyping

We begin with the basic rules of subtyping: reflexivity, transitivity, and subsumption. It is also convenient to add a type constant, Top, that is a supertype of every type. The judgment $E \vdash A <: B$ asserts that A is a subtype of B in environment E.

$\Delta_{<:}$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

(Val Subsumption)
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

We also allow specific subtyping relations between built-in algebras, such as Nat<:Int. We express the built-in subtypings by a given relation SubSort between types K∈Sort:

$\Delta_{<:K}$

(Sub Sort)
$$\frac{E \vdash \diamond \quad \langle K,K' \rangle \in SubSort}{E \vdash K <: K'}$$

After these general preliminaries, we write the subtyping rules for function and object types:

$\Delta_{<:\to}$

(Sub Arrow)

$$\frac{E \vdash A' <: A \qquad E \vdash B <: B'}{E \vdash A \to B <: A' \to B'}$$

$\Delta_{\mathbf{<:Ob}}$

(Sub Object)

$$\frac{E \vdash B_i \qquad \forall i \in 1..n+m}{E \vdash [l_i:B_i \ ^{i\in 1..n+m}] <: [l_i:B_i \ ^{i\in 1..n}]} \qquad (l_i \text{ distinct})$$

The subtyping rule for function types is the standard one. A function type $A \to B$ is *contravariant* in its domain, so $A \to B <: A' \to B$ requires $A' <: A$, and is *covariant* in its codomain, so $A \to B <: A \to B'$ requires $B <: B'$.

The subtyping rule for objects allows a longer object type $[l_i:B_i \ ^{i\in 1..n+m}]$ to be a subtype of a shorter object type $[l_i:B_i \ ^{i\in 1..n}]$. Moreover, an object type is *invariant* in its component types: $[l_i:B_i \ ^{i\in 1..n+m}] <: [l_i:B_i' \ ^{i\in 1..n}]$ requires $B_i \equiv B_i'$ for all $i \in 1..n$. That is, object types are neither covariant nor contravariant; in particular, $[l:A \to B]$ is neither covariant in B nor contravariant in A. The necessity of this invariance condition is explained in section 4.5.

We define the calculi:

$$\mathbf{Ob_{1<:}} \triangleq \mathbf{Ob_1} \cup \Delta_{<:} \cup \Delta_{\mathbf{<:K}} \cup \Delta_{\mathbf{<:Ob}}$$
$$\mathbf{F_{1<:}} \triangleq \mathbf{F_1} \cup \Delta_{<:} \cup \Delta_{\mathbf{<:K}} \cup \Delta_{<:\to}$$
$$\mathbf{FOb_{1<:}} \triangleq \mathbf{FOb_1} \cup \Delta_{<:} \cup \Delta_{\mathbf{<:K}} \cup \Delta_{\mathbf{<:Ob}} \cup \Delta_{<:\to}$$

Note that the translation of $\mathbf{F_1}$ into $\mathbf{Ob_1}$ does not extend to a corresponding translation of $\mathbf{F_{1<:}}$ into $\mathbf{Ob_{1<:}}$, because $\langle\!\langle A \to B \rangle\!\rangle = [\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:\langle\!\langle B \rangle\!\rangle]$ is invariant in $\langle\!\langle A \rangle\!\rangle$ and $\langle\!\langle B \rangle\!\rangle$. Hence, $\mathbf{Ob_{1<:}}$ is essentially a restricted version of $\mathbf{FOb_{1<:}}$ with invariant function types. We recover the covariant/contravariant subtyping properties of function types in section 6.2, by an encoding based on bounded universal and existential types.

4.1.1 Minimum Types

With the addition of subsumption we have obviously lost the unique-types property of $\mathbf{Ob_1}$ (see section 3.2.1). However, a weaker property holds: every term of $\mathbf{Ob_{1<:}}$ has a minimum type. This property also holds for $\mathbf{F_{1<:}}$, and we believe that it holds for $\mathbf{FOb_{1<:}}$.

We consider a system $\mathbf{MinOb_{1<:}}$ obtained from $\mathbf{Ob_{1<:}}$ by removing (Val Subsumption), and by modifying the (Val Object) and (Val Override) rules as follows:

(Val Min Object) (where $A \equiv [l_i:B_i \ ^{i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i : B_i' \qquad E \vdash B_i' <: B_i \qquad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i:A)b_i \ ^{i\in 1..n}] : A}$$

(Val Min Override) (where $A \equiv [l_i:B_i \ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A' \qquad E \vdash A' <: A \qquad E, x:A \vdash b : B_j' \qquad E \vdash B_j' <: B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b : A}$$

Typing in $\mathbf{MinOb}_{1<:}$ is unique, as we show below. We can easily extract from $\mathbf{MinOb}_{1<:}$ a type-checking algorithm that, given an environment E and a term a, computes a type A such that $E \vdash a : A$ if one exists. The algorithm would have to call a decision procedure for the subsort relation.

The next three propositions are proved by easy inductions on the derivations of $E \vdash a : A$.

Proposition 4.1.1-1 ($\mathbf{MinOb}_{1<:}$ typings are $\mathbf{Ob}_{1<:}$ typings)

If $E \vdash a : A$ is derivable in $\mathbf{MinOb}_{1<:}$, then it is also derivable in $\mathbf{Ob}_{1<:}$.

□

Proposition 4.1.1-2 ($\mathbf{MinOb}_{1<:}$ has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{MinOb}_{1<:}$, then $A \equiv A'$.

□

Proposition 4.1.1-3 ($\mathbf{MinOb}_{1<:}$ has smaller types than $\mathbf{Ob}_{1<:}$)

If $E \vdash a : A$ is derivable in $\mathbf{Ob}_{1<:}$ then $E \vdash a : A'$ is derivable in $\mathbf{MinOb}_{1<:}$ for some A' such that $E \vdash A'<:A$ is derivable (in either system).

□

Proposition 4.1.1-4 ($\mathbf{Ob}_{1<:}$ has minimum types)

In $\mathbf{Ob}_{1<:}$, if $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and, for any A', if $E \vdash a : A'$ then $E \vdash B<:A'$.

Proof

Assume $E \vdash a : A$. By Proposition 4.1.1-3, $E \vdash a : B$ is derivable in $\mathbf{MinOb}_{1<:}$ for some B such that $E \vdash B<:A$. By Proposition 4.1.1-1, $E \vdash a : B$ is also derivable in $\mathbf{Ob}_{1<:}$. If $E \vdash a : A'$, then $E \vdash a : B'$ is also derivable in $\mathbf{MinOb}_{1<:}$ for some B' such that $E \vdash B'<:A'$. By Proposition 4.1.1-2, $B \equiv B'$, so $E \vdash B<:A'$.

□

Just like lack of annotations for ς-binders destroys the unique-types property for $\mathbf{Ob}_1$, it destroys the minimum-types property for $\mathbf{Ob}_{1<:}$. For example, let:

$A = [l: []]$

$A' = [l: A]$

$a = [l = \varsigma(x)[l = \varsigma(x)[]]]$

then:

$\emptyset \vdash a:A \quad \text{and} \quad \emptyset \vdash a:A'$

but A and A' have no common subtype. This example also shows that minimum typing is lost for objects with fields (where the ς-binders are omitted entirely), since a can be written as $[l = [l = []]]$ with our conventions.

The term a.l:=[] typechecks using a:A but not using a:A'. Naive type inference algorithms might find the type A' for a, and fail to find any type for a.l:=[]. Thus, the absence of minimum typings poses practical problems for type inference. Palsberg has described an ingenious type inference algorithm that surmounts these problems [Palsberg 1993].

In contrast, in $\mathbf{Ob}_{1<:}$ (with annotations), $[l=\varsigma(x:A)[l=\varsigma(x:A)[]]] : A$ and $[l=\varsigma(x:A')[l=\varsigma(x:A)[]]] : A'$ are minimum typings.

4.2 Programming in $\mathbf{Ob_{1<:}}$

Both $\mathbf{Ob_{1<:}}$ and $\mathbf{FOb_{1<:}}$ are fairly useful object calculi with subsumption. In this section we show how to use them to express some common examples and concepts.

We begin by defining typed versions of one-dimensional and two-dimensional points:

P_x	$\triangleq$	[x:Real]	one-dimensional points on the x axis
P_y	$\triangleq$	[y:Real]	one-dimensional points on the y axis
P_{xy}	$\triangleq$	[x, y:Real]	two-dimensional points

We obtain a multiple-subtyping situation: $P_{xy} <: P_x$ and $P_{xy} <: P_y$.

We can enrich this example with proper methods, by extending two-dimensional points with polar coordinates:

P_{xyrt}	$\triangleq$	[x,y, r, t: Real] two-dimensional points with redundant coordinates
p	$\triangleq$	[x=0, y=1, r=ς(s:P_{xyrt})sqrt(s.x^2+s.y^2), t=ς(s:P_{xyrt})atan2(s.y,s.x)]
to-polar	$\triangleq$	λ(o:P_{xyrt})[x=ς(s:P_{xyrt})cos(s.t)*s.r, y=ς(s:P_{xyrt})sin(s.t)*s.r, r=o.r, t=o.t]
to-cart	$\triangleq$	λ(o:P_{xyrt})[x=o.x, y=o.y, r=ς(s:P_{xyrt})sqrt(s.x^2+s.y^2), t=ς(s:P_{xyrt})atan2(s.y,s.x)]

In p the cartesian coordinates are primitive, and the polar coordinates are computed by methods. The function to-polar maps a point to itself, except that the polar coordinates become primitive and the cartesian coordinates become derived; to-cart works in the opposite direction. Thus, to-polar and to-cart convert passive data to active computation, and vice versa.

We may want to calculate to-polar(p) before performing computations on p that are more efficient in polar representation. However, from the point of view of computing a correct result, there is no harm in not knowing which representation is primitive for a point, as long as we maintain the invariant that the cartesian and the polar coordinates represent the same point.

Since $P_{xyrt} <: P_{xy}$ we can pass a P_{xyrt} element q to a client that operates only with the cartesian coordinates. If we pass to-cart(q) instead of q, we can be certain that client updates to x and y will not break the representation invariant.

4.3 Classes and Inheritance

The object-oriented notions of *class* and *inheritance* are not explicit in our calculi. Here, we discuss how these notions can be represented.

4.3.1 Classes as Collections of Pre-Methods

Our discussion is rather informal and relies on common object-oriented jargon. To avoid ambiguity, we call pre-methods those functions that become methods once embedded into objects.

We take the point of view that "inheritance" means pre-method reuse, and that "classes" are collections of interdependent reusable pre-methods. As in Modula-3 [Nelson 1991], we make methods reusable by writing them first as functions (that is, as pre-methods), and then by repeatedly embedding these functions into objects.

The key idea is that if $A \equiv [l_i:B_i\ ^{i\in 1..n}]$ is an object type, then:

$$\text{Class}(A) \triangleq [l_i:A\to B_i\ ^{i\in 1..n}]$$

can be seen as a class type. A class (that is, a member of a class type) is an object that groups the pre-methods. The pre-methods in a class are fields that can be extracted to form objects of type A, and reused to form other classes. We shall see later an interesting application of proper methods within classes.

An incomplete class type $[l_i:A\to B_i\ ^{i\in 1..m}]$, for m<n, can be seen as an abstract-class type. An abstract class, belonging to such a type, must be completed with all the missing pre-methods before it can be instantiated.

We would like to say that a class Class(A) with more pre-methods inherits (or may inherit) from a class Class(A') with fewer pre-methods, possibly reusing some pre-methods of Class(A'). However, it can be seen easily that A<:A' does not imply Class(A)<:Class(A'): inheritance is not subtyping [Cook, Hill, Canning 1990]. (Neither does it imply Class(A')<:Class(A).) Therefore, we define an ad-hoc inheritance relation $\leq$ on class types that captures the intuition of method reuse. We set:

$$\text{Class(A)} \leq \text{Class(A')} \quad \text{iff} \quad A <: A' \qquad (\leq \text{ means "may inherit from"})$$

where we must have $A\equiv[l_i:B_i\ ^{i\in 1..n+m}]$ and $A'\equiv[l_i:B_i\ ^{i\in 1..n}]$.

If Class(A) $\leq$ Class(A'), then $A'\to B_i <: A\to B_i$, because of the contravariance of function types. Hence, pre-methods of c':Class(A') may be reused in assembling c:Class(A), by subsumption. Whenever c is defined by reassembling, extending, and modifying c', we may informally say that c is a subclass of c'.

The multiple subtyping property, which holds for object types, induces multiple inheritance on class types: a class can reuse pre-methods from any of its superclasses.

Given a class of type Class(A), it is easy to write a function new_A : Class(A) $\to$ A that generates objects from that class.

$$\text{new}_A \triangleq \lambda(c:\text{Class(A)})\ [l_i=\varsigma(s:A)c.l_i(s)\ ^{i\in 1..n}]$$

Further, we can define classes and class types with new built-in:

$$\begin{aligned}\text{Class}^{\text{new}}(A) &\triangleq [\text{new:A}, l_i:A\to B_i\ ^{i\in 1..n}]\\ \text{class}_A{}^{\text{new}} &\triangleq [\text{new}=\varsigma(c:\text{Class}^{\text{new}}(A))\ [l_i=\varsigma(s:A)c.l_i(s)\ ^{i\in 1..n}],\\ &\qquad l_i=\lambda(s:A)b_i\{s\}\ ^{i\in 1..n}]\end{aligned}$$

Here new is a proper method of $\text{class}_A{}^{\text{new}}$ that uses the other components of the class, through self, to generate new objects:

$$\text{class}_A{}^{\text{new}}.\text{new} : A$$

By subsumption, we may hide some or all of the component of the class. The components that are not hidden can be overridden; objects created from the resulting class will incorporate the overridden methods.

$$\begin{aligned}&\text{class}_A{}^{\text{new}} : [\text{new:A}, l_i:A\to B_i\ ^{i\in 1..k}] && k<n\\ &(\text{class}_A{}^{\text{new}}.l_j:=f).\text{new} : A && \text{where } f:A\to B_j \text{ is an overriding method}\end{aligned}$$

The self of a class is reminiscent of the notion of *super* in Smalltalk [Goldberg, Robson 1983]. If we write classes as follows:

$$\begin{aligned}\text{class}_A{}^{\text{new}} \triangleq [\ &\text{new= } \textit{(as previously)},\\ &l_i=\varsigma(c:\text{Class}^{\text{new}}(A))\ \lambda(s:A)b_i\{c,s\}\ ^{i\in 1..n}]\end{aligned}$$

then the pre-methods of a class are available within the methods of objects generated from the class.

4.3.2 Programming with Classes

As an example of this general approach to classes, consider one-dimensional points (P_{xon}) and two-dimensional points (P_{xyon}) with a norm method (n) and a test-for-origin method (o). We have $P_{xyon} <: P_{xon}$:

$P_{xon} \triangleq [x: Real, o: Bool, n: PosReal]$

$P_{xyon} \triangleq [x, y: Real, o: Bool, n: PosReal]$

The class types corresponding to these objects types are:

$Class(P_{xon}) \triangleq [x_{pre}: P_{xon} \to Real,\ o_{pre}: P_{xon} \to Bool,\ n_{pre}: P_{xon} \to PosReal]$

$Class(P_{xyon}) \triangleq [x_{pre}, y_{pre}: P_{xyon} \to Real,\ o_{pre}: P_{xyon} \to Bool,\ n_{pre}: P_{xyon} \to PosReal]$

The pre-method signatures have an explicit self parameter, P_{xon} or P_{xyon}.

Since x and y are intended to be instance variables, we may decide to omit x_{pre} and y_{pre} from the class types (in a slight variation on the general treatment of classes). The values for these instance variables will be provided at object creation. Hence we define:

$ClassP_{xon} \triangleq [o_{pre}: P_{xon} \to Bool,\ n_{pre}: P_{xon} \to PosReal]$

$ClassP_{xyon} \triangleq [o_{pre}: P_{xyon} \to Bool,\ n_{pre}: P_{xyon} \to PosReal]$

Next, we define class values as collections of functions with explicit self parameters.

$classP_{xon} : ClassP_{xon} \triangleq$

$[o_{pre} = \lambda(p:P_{xon})\ p.n =_{Real} 0.0,$

$n_{pre} = \lambda(p:P_{xon})\ abs(p.x)]$

The o_{pre} method of class $classP_{xon}$ is defined from the n component of P_{xon}; this is an implementation choice that creates a dependency between methods in the class. In fact, this is a delayed dependency, in the style of *mixins* [Weinreb, Moon 1981], because p.n is not yet bound to n_{pre}.

We now define the subclass $classP_{xyon}$ of $classP_{xon}$. The method $classP_{xon}.o_{pre}$ is reused (inherited). The n_{pre} method is instead redefined (overridden) as a two-dimensional norm.

$classP_{xyon} : ClassP_{xyon} \triangleq$

$[o_{pre} = classP_{xon}.o_{pre},$

$n_{pre} = \lambda(p:P_{xyon})\ sqrt(p.x\text{^}2 * p.y\text{^}2)]$

Finally we create object instances of each class by combining values for the instance variables with a method suite provided by the class. The code for generating instances is uniform for every class, once the instance variables are provided, and consists of applying every pre-method to the object's self.

$newP_{xon} : Real \to ClassP_{xon} \to P_{xon} \triangleq$

$\lambda(x':Real)\ \lambda(c:ClassP_{xon})$

$[x = x',\ o = \varsigma(s:P_{xon})\ c.o_{pre}(s),\ n = \varsigma(s:P_{xon})\ c.n_{pre}(s)]$

$newP_{xon}(3.0)(classP_{xon}) =$

$[x = 3.0,$

$o = \varsigma(p:P_{xon})\ p.n =_{Real} 0.0,$

$n = \varsigma(p:P_{xon})\ abs(p.x)]$

$newP_{xyon} : Real \to Real \to ClassP_{xyon} \to P_{xyon} \triangleq$ ...

Similar results can be obtained by putting new inside classes. The main difference is in the handling of instance variables. We can hide all the components of a class except new and chosen instance variables. A simple call to new will create an object where the instance variables receive default values. Alternatively, we can override the instance variables before calling new.

Many variations on this theme are possible. An actual programming language with classes, based on these techniques, would have to cast into syntax some carefully chosen idioms.

4.4 Equational Theory of $\mathbf{Ob_{1<:}}$

We extend the equational theory of $\mathbf{Ob_1}$ to take subsumption into account. First, the following equalities are associated with the $\Delta_{<:}$ fragment:

$\Delta_{=<:}$

(Eq Subsumption)
$$\frac{E \vdash a \leftrightarrow a' : A \qquad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Top)
$$\frac{E \vdash a:A \qquad E \vdash b:B}{E \vdash a \leftrightarrow b : Top}$$

Still, this does not give us enough power to compare objects of different lengths (except as members of Top). We remedy this by augmenting the $\Delta_{=Ob}$ fragment from section 3.3 with the rule (Eq Sub Object), which allows us to ignore methods that do not appear in the type of an object. At the same time we generalize (Eval Select) and (Eval Override) to deal with objects and types of different lengths:

$\Delta_{=Ob<:}$

(Eq Sub Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $A' \equiv [l_i:B_i\ ^{i\in 1..n}, l_j:B_j\ ^{j\in n+1..m}]$)
$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \qquad E, x_j:A' \vdash b_j : B_j \quad \forall j \in n+1..m}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \leftrightarrow [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}] : A}$$

(Eval Select) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)
$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)
$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \leftrightarrow [l_j=\varsigma(x:A')b, l_i=\varsigma(x_i:A')b_i\ ^{i\in (1..n+m)-\{j\}}] : A}$$

According to (Eq Sub Object) an object can be truncated to its externally visible collection of methods, but only if those methods do not depend on the hidden ones. (The truncated object would not work otherwise.) It is hard to obtain more than this, as we now discuss.

In section 3.3 we saw that a and b below cannot be equal at type A:

$$\begin{aligned} A &\triangleq [x:Nat, f:Nat] \\ a:A &\triangleq [x=1, f=\varsigma(s:A)1] \\ b:A &\triangleq [x=1, f=\varsigma(s:A)s.x] \end{aligned}$$

Using (Eq Sub Object) it is at least possible to show that a and b are equal at type [x:Nat], by showing $\emptyset \vdash a \leftrightarrow [x=1]:[x:Nat]$ and $\emptyset \vdash b \leftrightarrow [x=1]:[x:Nat]$.

We can ask next whether a and b are equal at type [f:Nat]. This seems reasonable because the only way to distinguish a and b is to update their x components, which are not exposed in [f:Nat]. However, we cannot apply (Eq Object) and (Eq Sub Object) because we would need to show $s:[f:Nat] \vdash 1 \leftrightarrow s.x : Nat$.

A stronger rule would be required to show $\emptyset \vdash a \leftrightarrow b:[f:Nat]$. Suppose, in general, that c is an object with a visible method $p=\varsigma(s:[p:P,q:Q])p'$ and a hidden method $q=\varsigma(s:[p:P,q:Q])q'$. We might try to replace any occurrence of s.q in p' with the current value of c.q. This, however, would not give an equivalent object because q' may refer to a visible component of c that may be updated later. Hence we must replace s.q in p' literally with q' (with self appropriately renamed, if necessary).

There are however at least two problems with this idea. First, if q' contains occurrences of s.q, then $p'\{s.q \leftarrow q'\}$ still contains occurrences of s.q, and it is not clear how to go further. Second, unlike all the rest of the equational theory, the transformation $p'\{s.q \leftarrow q'\}$ (as well as $\emptyset \vdash a \leftrightarrow b:[f:Nat]$) is invalid in an imperative language where somebody might hold a pointer to the whole object, c, and modify methods that are hidden to us but not to them. We leave as an open issue the status of equations such as $\emptyset \vdash a \leftrightarrow b:[f:Nat]$.

4.5 Objects Versus Records

We consider two natural but incorrect extensions of $\mathbf{Ob}_{1<:}$: covariant object types and method extraction. Taken together, these two failed extensions show that, in a calculus with subsumption, object typing is fundamentally different from record typing since both extensions are sound for records. A third extension, elder, allows us to define a new version of a method in terms of the previous one.

4.5.1 Covariant Object Types

The subtyping rule for object types should be compared with the analogous rule for cartesian products. Object types are invariant in their components, while cartesian products are covariant in both of their components. Similarly, record types $\langle l_i:A_i \ ^{i \in 1..n} \rangle$, which generalize cartesian products, are covariant in all of their components.

It is natural to ask what happens if we allow object types to be covariant in their components. Suppose we adopt the following more liberal subtyping rule for objects:

(Sub Object/covariant)

$$\frac{E \vdash B_i <: B_i' \quad E \vdash B_j \quad \forall i \in 1..n,\ j \in n+1..m}{E \vdash [l_i:B_i \ ^{i \in 1..n+m}] <: [l_i:B_i' \ ^{i \in 1..n}]} \qquad (l_i \text{ distinct})$$

Then we can derive a contradiction. Assume PosReal<:Real and ln:PosReal→Real (the natural logarithm). Define:

P	≜	[x:Real, f:Real]	note that [x=1.0, f=ς(s:P)ln(s.x)] : P is not derivable
Q	≜	[x:PosReal, f:Real]	we have Q <: P by (Sub Object/covariant)
a	≜	[x=1.0, f=ς(s:Q)ln(s.x)]	we have a:Q by (Val Object), hence also a:P by (Val Subsumption)
b	≜	a.x:= -1.0	from a:P we have b:P by (Val Override), therefore b.f:Real

Now we have $\emptyset \vdash b.f \leftrightarrow \ln(-1.0) : Real$; we can derive a typing for a program that applies a function to an argument outside its domain. Hence, the type system is unsound, as a direct consequence of adding (Sub Object/covariant). The above example can be recast with other nontrivial subtypings in place of PosReal<:Real, such as a subtyping between object types. Note that the example involves simple field update, and does not require proper method override.

The basic reason for this problem is that each method relies on the types of the other methods, through the type of self. This dependence is essentially contravariant, and hence is incompatible with covariance.

4.5.2 Method Extraction

Let us now assume we have an operation for extracting a method from an object, as discussed at the beginning of section 4:

$$\text{(Val Extract)} \qquad (\text{where } A \equiv [l_i{:}B_i \ ^{i \in 1..n}]) \qquad \frac{E \vdash a : A \quad j \in 1..n}{E \vdash a \cdot l_j : A \to B_j}$$

$$\text{(Eval Extract)} \qquad (\text{where } A \equiv [l_i{:}B_i \ ^{i \in 1..n}],\ a \equiv [l_i = \varsigma(x_i{:}A')b_i \ ^{i \in 1..n+m}]) \qquad \frac{E \vdash a : A \quad j \in 1..n}{E \vdash a \cdot l_j \ \leftrightarrow \ \lambda(x_j{:}A)b_j : A \to B_j}$$

Then we define:

P	≜	[x:Int, f:Int]	
p	≜	[x=1, f=1]	we have p:P by (Val Object)
Q	≜	[x,y:Int, f:Int]	we have Q <: P by (Sub Object)
a	≜	[x=1, y=1, f=ς(s:Q)s.x+s.y]	we have a:Q by (Val Object), hence also a:P by (Val Subsumption)
b	≜	a·f	we have b:P→Int by (Val Extract), with b ↔ λ(s:P)s.x+s.y : P→Int by (Eval Extract)

Now we have $\emptyset \vdash b(p) \leftrightarrow (\lambda(s{:}P)s.x+s.y)(p) \leftrightarrow p.x+p.y : \text{Int}$, via (Eval Beta), but p does not have a y component. If we change an A to A' in the conclusion of (Eval Extract), which becomes $E \vdash a \cdot l_j \ \leftrightarrow \ \lambda(x_j{:}A')b_j : A \to B_j$, then the (Eval Beta) step is not even possible.

Hence, it is unsound to add the method extraction operation to $\mathbf{FOb_{1<:}}$ (or to $\mathbf{Ob_{1<:}}$, with encoded function types), although it is sound to add it to $\mathbf{FOb_1}$ (or to $\mathbf{Ob_1}$).

4.5.3 Elder

Although in general it is unsound to extract a method, it is sound to refer to the previous value of a method in the course of an override. Just like the new value of a record field can be defined from its old value, the new code for a method can reuse the overridden code. We write $a.l_j \Leftarrow \varsigma(x{:}A,y{:}B_j)b$ for the result of overriding the l_j method of a with $\varsigma(x{:}A,y{:}B_j)b$; when l_j is invoked, x is self and y is the body of the old method. We call y *elder*, because of the similarity with the inner construct of Simula-67 [Dahl, Nygaard 1966].

We give only the type and evaluation rules for override with elder:

$$\text{(Val Override with elder)} \qquad (\text{where } A \equiv [l_i{:}B_i \ ^{i \in 1..n}]) \qquad \frac{E \vdash a : A \quad E, x{:}A, y{:}B_j \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A,y{:}B_j)b : A}$$

$$\text{(Eval Override with elder)} \qquad (\text{where } A \equiv [l_i{:}B_i \ ^{i \in 1..n}],\ a \equiv [l_i = \varsigma(x_i{:}A')b_i \ ^{i \in 1..n+m}],\ x_j \notin \text{dom}(E)) \qquad \frac{E \vdash a : A \quad E, x{:}A, y{:}B_j \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A,y{:}B_j)b\{x,y\} \ \leftrightarrow \ [l_j = \varsigma(x_j{:}A')b\{x_j,b_j\}, l_i = \varsigma(x_i{:}A')b_i \ ^{i \in (1..n+m)-\{j\}}] : A}$$

Override with elder could easily be implemented from method extraction. But, unlike method extraction, override with elder is sound because we never apply the old method to an arbitrary element of type A. The old method's self remains bound to the object's self, of type A'.

In a language like ours, with primitive objects but without primitive classes, this mechanism provides a form of inheritance by reusing methods of existing objects in new objects. We describe a typical example. Consider a two-dimensional point p with a draw method that produces a picture with the point in it. This point would have the type $P_{xyd} \triangleq [x,y:Real, draw:Bitmap]$. We may change the draw method in order to invert the color of the picture, reusing the existing drawing code: $p.draw \Leftarrow \varsigma(s:P_{xyd},e:Bitmap)invert(e)$.

For the sake of simplicity, we do not add elder to our core calculus. However, we think this should be a useful construct in practice. In section 8.2.3 we show a technique that may achieve the same effect as override with elder.

5. First-Order Calculi with Recursion

In $\mathbf{Ob_1}$ we can find typings for objects that use or modify self, such as $[l = \varsigma(x:[l:[]])\ x.l] : [l:[]]$. However, we cannot find good typings for objects whose methods return either self or an updated self, such as the movable points of section 2.4.1: this feature calls for the use of recursive types.

In this section we discuss recursion in first-order calculi with and without subtyping, and we provide typings for some of the untyped examples of section 2.4.

5.1 Recursion

Recursive types and values can be added via standard formal system fragments. However, we need to assume an appropriate syntactic criterion $A \succ X$ ("A formally contractive in X"), dependent on a whole calculus. The predicate $A \succ X$ must be sufficient to establish the uniqueness of fixpoints [MacQueen, Plotkin, Sethi 1986; Amadio, Cardelli 1991]; typically A cannot be simply X or $\mu(Y)X$ (see appendix E).

Δ_X

(Env X)
$$\frac{E \vdash \diamond \qquad X \notin dom(E)}{E,X \vdash \diamond}$$

(Type X)
$$\frac{E',X,E'' \vdash \diamond}{E',X,E'' \vdash X}$$

Δ_μ

(Type Rec)
$$\frac{E,X \vdash A \qquad A \succ X}{E \vdash \mu(X)A}$$

(Val Fold)
$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)
$$\frac{E \vdash a : \mu(X)A}{E \vdash unfold(a) : A\{X \leftarrow \mu(X)A\}}$$

(Val Rec)

$$\frac{E,x{:}A \vdash a : A}{E \vdash \mu(x{:}A)a : A}$$

We obtain the calculi:

$\mathbf{Ob}_{1\mu}$	$\triangleq$	$\mathbf{Ob}_1 \cup \Delta_X \cup \Delta_\mu$	$\mathbf{Ob}_1$ with recursion
$\mathbf{F}_{1\mu}$	$\triangleq$	$\mathbf{F}_1 \cup \Delta_X \cup \Delta_\mu$	$\mathbf{F}_1$ with recursion
$\mathbf{FOb}_{1\mu}$	$\triangleq$	$\mathbf{FOb}_1 \cup \Delta_X \cup \Delta_\mu$	$\mathbf{FOb}_1$ with recursion

and equations:

$\Delta_{=\mu}$

(Eq Fold)

$$\frac{E \vdash a \leftrightarrow a' : A\{X\leftarrow\mu(X)A\}}{E \vdash \mathrm{fold}(\mu(X)A, a) \leftrightarrow \mathrm{fold}(\mu(X)A, a') : \mu(X)A}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash \mathrm{unfold}(a) \leftrightarrow \mathrm{unfold}(a') : A\{X\leftarrow\mu(X)A\}}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash \mathrm{fold}(\mu(X)A,\mathrm{unfold}(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X\leftarrow\mu(X)A\}}{E \vdash \mathrm{unfold}(\mathrm{fold}(\mu(X)A,a)) \leftrightarrow a : A\{X\leftarrow\mu(X)A\}}$$

(Eq Rec)

$$\frac{E,x{:}A \vdash a \leftrightarrow a' : A}{E \vdash \mu(x{:}A)a \leftrightarrow \mu(x{:}A)a' : A}$$

(Eval Rec)

$$\frac{E,x{:}A \vdash a : A}{E \vdash \mu(x{:}A)a \leftrightarrow a\{x\leftarrow\mu(x{:}A)a\} : A}$$

In this formulation of recursive types, the isomorphism between $\mu(X)A$ and $A\{X\leftarrow\mu(X)A\}$ is explicit; the fold/unfold coercions must be used explicitly to move around the isomorphism. The types $\mu(X)A$ and $A\{X\leftarrow\mu(X)A\}$ are not equivalent, so we have no need for an additional type equivalence judgment. See [Amadio, Cardelli 1991] for a formulation of recursive types where the isomorphism is implicit.

As an example, these rules yield that, if $A \triangleq \mu(X)[l{:}X]$, then the self-returning object $a \triangleq \mathrm{fold}(A, [l=\varsigma(x{:}[l{:}A])\mathrm{fold}(A,x)])$ has type A, with $\mathrm{unfold}(a).l \leftrightarrow a$.

We can also write a typed version of the movable one-dimensional points of section 2.4.1:

$$P_1 \triangleq \mu(X)[x{:}\mathrm{Int}, \mathrm{mv_x}{:}\mathrm{Int}\rightarrow X]$$
$$\mathrm{origin}_1 \triangleq \mathrm{fold}(P_1,\ [x = 0,\ \ \mathrm{mv_x} = \varsigma(s{:}[x{:}\mathrm{Int}, \mathrm{mv_x}{:}\mathrm{Int}\rightarrow P_1])\ \lambda(dx{:}\mathrm{Int})\ s.x{:=}s.x{+}dx])$$

Taking advantage of recursive types, we can give a translation for the untyped λ-calculus in $\mathbf{Ob}_{1\mu}$, along the lines of the translation of the untyped λ-calculus in section 2.2. The new translation does not use the (Val Rec) rule; type recursion is sufficient:

Typed translation of the untyped λ-calculus

$\Lambda \triangleq \mu(X)[\mathrm{arg}{:}X, \mathrm{val}{:}X]$	(globally free variables should be assigned type Λ)
$\langle\!\langle x \rangle\!\rangle_\rho \triangleq \rho(x)$	$\rho \in \mathrm{Var} \rightarrow \mathbf{Ob}_{1\mu}$-term

$$\langle\!\langle b(a) \rangle\!\rangle_\rho \triangleq (\text{unfold}(\langle\!\langle b \rangle\!\rangle_\rho).\text{arg} \Leftarrow \varsigma(x{:}\Lambda)\ \langle\!\langle a \rangle\!\rangle_\rho).\text{val} \quad \text{for } x \notin FV(\langle\!\langle a \rangle\!\rangle_\rho)$$

$$\emptyset(x) \triangleq x \qquad (\rho\{y \leftarrow a\})(x) \triangleq \text{if } x{=}y \text{ then } a \text{ else } \rho(x)$$

$$\langle\!\langle \lambda(x)b \rangle\!\rangle_\rho \triangleq \text{fold}(\Lambda, [\text{arg} = \varsigma(x{:}\Lambda)\ \text{unfold}(x).\text{arg},\ \text{val} = \varsigma(x{:}\Lambda)\ \langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow \text{unfold}(x).\text{arg}\}}])$$

5.2 Recursion and Subsumption

To add recursion to a calculus with subtyping we cannot reuse directly the Δ_μ fragment of section 5.1, because type variables in environments now require subtype bounds. These bounded variables are needed in a new typing rule for subtyping recursive types [Amadio, Cardelli 1991]. We still write E,X,E', but now as an abbreviation for E,X<:Top,E'.

$\Delta_{<:X}$

(Env X<:) $\dfrac{E \vdash A \quad X \notin \text{dom}(E)}{E,X{<:}A \vdash \diamond}$ (Type X<:) $\dfrac{E',X{<:}A,E'' \vdash \diamond}{E',X{<:}A,E'' \vdash X}$ (Sub X) $\dfrac{E',X{<:}A,E'' \vdash \diamond}{E',X{<:}A,E'' \vdash X{<:}A}$

$\Delta_{<:\mu}$

(Type Rec<:) $\dfrac{E,X \vdash A \quad A > X}{E \vdash \mu(X)A}$ (Sub Rec) $\dfrac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y,X{<:}Y \vdash A{<:}B}{E \vdash \mu(X)A <: \mu(Y)B}$

(Val Fold) $\dfrac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) : \mu(X)A}$ (Val Unfold) $\dfrac{E \vdash a : \mu(X)A}{E \vdash \text{unfold}(a) : A\{X \leftarrow \mu(X)A\}}$ (Val Rec) $\dfrac{E,x{:}A \vdash a : A}{E \vdash \mu(x{:}A)a : A}$

$\Delta_{=<:\mu} \triangleq \Delta_{=\mu}$ plus the following generalization of (Eq Fold)

(Eq Fold<:) $\dfrac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y,X{<:}Y \vdash A{<:}B \quad E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) \leftrightarrow \text{fold}(\mu(Y)B, a') : \mu(Y)B}$

The (Sub Rec) rule determines the variance behavior of recursive types. If A{X,Y} is covariant in X, then the variance of Y in $\mu(X)A\{X,Y\}$ is the same as the variance of Y if A{X,Y}. But if A{X,Y} is contravariant X, then $\mu(X)A\{X,Y\}$ is always invariant in Y (because after unfolding we obtain $A\{\mu(X)A\{X,Y\},Y\}$ with Y in positions of opposite variance). Similarly, if A{X,Y} is invariant in X then $\mu(X)A\{X,Y\}$ is always invariant in Y.

The (Eq Fold<:) rule is a generalization of (Eq Fold) that can prove the equality of two fold terms even when the types used for folding are different. The hypotheses imply a subtype relation between

the types used for folding: $\mu(X)A <: \mu(Y)B$. A more flexible congruence rule is derivable from (Eq Fold<:):

(Eq Fold<: Lemma)

$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y,X<:Y \vdash A<:B \quad E \vdash a \leftrightarrow a' : B\{Y \leftarrow \mu(Y)B\} \quad E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \mathrm{fold}(\mu(X)A, a) \leftrightarrow \mathrm{fold}(\mu(Y)B, a') : \mu(Y)B}$$

For the derivation of (Eq Fold<: Lemma), we use (Eq Fold<:) twice, first with a≡a' and then with A≡B, and apply transitivity to the two results.

We now obtain the calculi:

$$\begin{aligned} \mathbf{Ob}_{\mathbf{1}<:\mu} &\triangleq \mathbf{Ob}_{\mathbf{1}<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mu} \\ \mathbf{F}_{\mathbf{1}<:\mu} &\triangleq \mathbf{F}_{\mathbf{1}<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mu} \\ \mathbf{FOb}_{\mathbf{1}<:\mu} &\triangleq \mathbf{FOb}_{\mathbf{1}<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mu} \end{aligned}$$

$\mathbf{FOb}_{\mathbf{1}<:\mu}$ is the strongest first-order system we consider in this paper, and $\mathbf{Ob}_{\mathbf{1}<:\mu}$ is a rather mild restriction of it. We complete section 5 by using these systems to provide typings for some of the examples from section 2.4, and by discussing a limitation of both $\mathbf{Ob}_{\mathbf{1}<:\mu}$ and $\mathbf{FOb}_{\mathbf{1}<:\mu}$.

5.3 Typing the Examples

In this section we provide first-order typings for the examples of sections 2.4.2, 2.4.3, and 2.4.4. We omit a discussion of the example of section 2.4.1. All the examples are revisited in section 7.2.

We start with the example of objects with backup; the type in question is:

$$\mathrm{Bk} \triangleq \mu(X)[\mathrm{retrieve}:X, \mathrm{backup}:X, ...]$$

We obtain the following typed version of the code, where $\mathrm{UBk} \triangleq [\mathrm{retrieve}:\mathrm{Bk}, \mathrm{backup}:\mathrm{Bk}, ...]$ is the unfolding of Bk:

$$\begin{aligned} &\mathrm{fold}(\mathrm{Bk}, \\ &\quad [\ \mathrm{retrieve} = \varsigma(s_1:\mathrm{UBk})\ \mathrm{fold}(\mathrm{Bk}, s_1), \\ &\quad\ \ \mathrm{backup} = \varsigma(s_2:\mathrm{UBk})\ \mathrm{fold}(\mathrm{Bk}, s_2.\mathrm{retrieve} \Leftarrow \varsigma(s_1:\mathrm{UBk})\ \mathrm{fold}(\mathrm{Bk}, s_2)), \\ &\quad\ \ ... \\ &\quad]) \ :\ \mathrm{Bk} \end{aligned}$$

If we now consider the types Point=[x,y:Int] and PointBk=μ(X)[retrieve:X, backup:X, x,y:Int], we obtain PointBk <: Point modulo an unfolding. Thus, points with backup can subsume points.

Similar techniques can be used to type the calculator example:

$$\mathrm{Calc} = \mu(X)[\mathrm{arg},\mathrm{acc}: \mathrm{Real}, \mathrm{enter}: \mathrm{Real} \to X, \mathrm{add},\mathrm{sub}: X, \mathrm{equals}: \mathrm{Real}]$$

Furthermore, we would like to obtain inclusions such as Calc <: μ(X)[enter: Real→X, add,sub: X, equals: Real], which would allow us to hide the implementation details of a calculator. Such inclusions are the subject of the next section.

The example of the numerals requires some additions to our first-order type system. One possibility is second-order types, to be studied later. A simpler, first-order alternative is sum types, if we are prepared to rephrase the example. We add a type construction A+B, with operations $\mathrm{inl}_{AB}:A\to(A+B)$, $\mathrm{inr}_{AB}:B\to(A+B)$, and $\mathrm{if}_{ABC}: (A+B)\to(A\to C)\to(B\to C)\to C$. (From now on we omit the subscripts.) Informally, A+B is the disjoint union of A and B, inl and inr are the obvious injections, and if is used to

examine elements of A+B returning elements of C. For convenience, we also add a type Unit with a constant unit:Unit.

The numerals can be expressed using sums as follows:

$$\text{Nat} \triangleq \mu(X)\ [\text{case: Unit+X, succ: X}]$$

$$\text{zero} \triangleq \text{fold(Nat, [case = inl(unit), succ = } \varsigma(x{:}[\text{case:Unit+Nat, succ:Nat}])\ \text{fold(Nat, x.case := inr(fold(Nat, x)))]}$$

$$\text{iszero} \triangleq \lambda(n{:}\text{Nat})\ \text{if(unfold(n).case)}(\lambda(u{:}\text{Unit})\ \text{true})(\lambda(p{:}\text{Nat})\ \text{false})$$

$$\text{pred} \triangleq \lambda(n{:}\text{Nat})\ \text{if(unfold(n).case)}(\lambda(u{:}\text{Unit})\ \text{zero})(\lambda(p{:}\text{Nat})\ p)$$

Although this code looks quite different from that of section 2.4.3, the two versions are related by a type isomorphism (explained in section 7.2.3).

5.4 The Shortcomings of $\mathbf{Ob_{1<:\mu}}$

The $\mathbf{Ob_{1<:\mu}}$ calculus looks very promising because it adds subtyping to a rich first-order theory. For example, we can use it to write types of movable points, as we did in section 5.1 with $\mathbf{Ob_{1\mu}}$:

P_1	$\triangleq$	$\mu(X)[x{:}\text{Int}, \text{mv_x:Int}\to X]$	movable one-dimensional points
P_2	$\triangleq$	$\mu(X)[x,y{:}\text{Int}, \text{mv_x,mv_y:Int}\to X]$	movable two-dimensional points

In addition, since we now have subtyping, we would expect to obtain that $P_2 <: P_1$, since intuitively P_2 extends P_1. However this is not provable, because the invariance of object types blocks the application of (Sub Rec) to the result type of mv_x.

Moreover, if we somehow allow $P_2 <: P_1$, we obtain an inconsistency. Briefly, suppose we use subsumption from $p{:}P_2$ to $p{:}P_1$, and then override the mv_x method of p with one that returns a proper element of P_1. Then, some other method of p may go wrong because it assumes that mv_x produces an element of P_2. More precisely, let us define:

UP_1	$\triangleq$	$[x{:}\text{Int}, \text{mv_x:Int}\to P_1]$	(the unfolding of P_1)
UP_2	$\triangleq$	$[x,y{:}\text{Int}, \text{mv_x,mv_y:Int}\to P_2]$	(the unfolding of P_2)

$$p_2 : P_2 \triangleq \text{fold}(P_2, [x=\varsigma(s_2{:}UP_2)\text{unfold}(s_2.\text{mv_x}(1)).y,\ y=0,\ \text{mv_x}=\varsigma(s_2{:}UP_2)\lambda(dx{:}\text{Int})\text{fold}(P_2,s_2),\ \text{mv_y}=\varsigma(s_2{:}UP_2)\lambda(dy{:}\text{Int})\text{fold}(P_2,s_2)])$$

$$p_1 : P_1 \triangleq \text{fold}(P_1, [x=0,\ \text{mv_x}=\varsigma(s_1{:}UP_1)\lambda(dx{:}\text{Int})\text{fold}(P_1,s_1)])$$

$$p : P_1 \triangleq p_2 \qquad \text{(retyping } p_2 \text{ using the assumption } P_2 <: P_1\text{)}$$

$$q : P_1 \triangleq \text{fold}(P_1, \text{unfold}(p).\text{mv_x} := \lambda(dx{:}\text{Int})p_1)$$

We have:

unfold(q).x

$\leftrightarrow$ (unfold(p_2).mv_x := λ(dx:Int)p_1).x

$\leftrightarrow$ [x=ς(s_2:P'$_2$)unfold(s_2.mv_x(1)).y, mv_x=λ(dx:Int)p_1, y=..., mv_y=...].x

$\leftrightarrow$ unfold(p_1).y

But unfold(p_1) does not have a y component.

As we have just seen, the failure of P_2<:P_1 is necessary. At the same time, it is unacceptable: in the common situation where a method returns an updated self, we lose all useful subsumption relations. The situation is less severe in imperative languages, where the mv_x method could be redefined to side-effect the host point and return nothing. Then, the type of the modified method would not depend on the type of self, and P_1 and P_2 would not be recursive. Even in imperative languages, though, we often find methods that, like mv_x, allocate new objects of the type of self and return them.

In many programming languages, such as Simula-67 and Modula-3 [Nelson 1991], the failure of P_2<:P_1 is avoided by not allowing a subclass to change the type of a method of a superclass. For example, in our calculus, we could define mv_x to return P_1 even when embedded in P_2:

$P_1 \triangleq \mu(X)[x:Int, mv_x:Int\rightarrow X]$

$P_2' \triangleq \mu(X)[x,y:Int, mv_x:Int\rightarrow P_1, mv_y:Int\rightarrow X]$

$UP_1 \triangleq [x:Int, mv_x:Int\rightarrow P_1]$ (the unfolding of P_1)

$UP_2' \triangleq [x,y:Int, mv_x:Int\rightarrow P_1, mv_y:Int\rightarrow P_2']$ (the unfolding of P_2')

Then we have UP_2' <: UP_1, so we can at least convert every point p:P_2' to a point fold(P_1,unfold(p)) of type P_1. It is possible to strengthen the type theory to identify recursive types up to isomorphism, as in [Amadio, Cardelli 1991]; then we can obtain directly P_2'<:P_1. Whenever we invoke mv_x on an element of P_2', though, we "forget" its second dimension. For this kind of solution to be useful, Simula-67 and Modula-3, among other languages, provide dynamic testing of membership in a subtype of a given type, so that the forgotten information can be recovered. In our example, we would test for membership in the subtype P_2' of P_1.

Hence, following the approach of common object-oriented languages, we could reasonably end this paper by adding *dynamic types* [Abadi, *et al.* 1991] to $\mathbf{Ob_{1<:\mu}}$. We would abandon the notion of truly static typing of subsumption. In the next section we start describing a better possibility.

5.5 Towards the Type Self

Some languages, such as Eiffel [Meyer 1988], have proposed a mechanism that aims to support relations such as P_2<:P_1 without requiring dynamic type testing. The proposal introduces a special type *Self* that denotes, in every subtype of an object type, the subtype in question. Using Self, we would write:

$P'_1 \triangleq [x:Int, mv_x:Int\rightarrow Self]$

$P'_2 \triangleq [x,y:Int, mv_x,mv_y:Int\rightarrow Self]$

assuming that we can establish the properties p'_2.mv_x(n):P'_2, p'_1.mv_x(n):P'_1, and P'_2<:P'_1.

The counterexample to P_2<:P_1 given earlier still looms. In fact, Eiffel has soundness problems because of such counterexamples [Cook 1989b]; these problems are addressed only outside the typing discipline.

Still, it may be possible to use the type Self, instead of recursive types, and to adopt ad-hoc typing rules for Self that avoid unsoundness. An appealing idea is to require that any term meant to override a

method returning Self should be "parametric in self " [Mitchell, Honsell, Fisher 1993]: that is, it should always return a modification of its self parameter, and never return a fixed object. This requirement blocks the counterexample in section 5.4, while allowing the definition of useful movable points.

The tentative conclusion we should draw here is that recursive types are the wrong way of modeling the type Self (whatever this is). We could now proceed to extend $\mathbf{Ob}_{\mathbf{1}<:\mu}$ with some ad-hoc notion of Self. However, this approach would involve a fair amount of "guessing" to find the right typing rules, which are not at all obvious. Instead, we now turn to second-order systems. We shall see that a plain second-order extension of $\mathbf{Ob}_{\mathbf{1}<:\mu}$ is sufficient for our purposes, essentially because there we can express the notion of being "parametric in self ".

6. Second-Order Calculi

In this section we present standard second-order extensions of first-order calculi. No new or unusual constructions are introduced. However, second-order quantifiers can be combined with recursive types to produce an interesting new concept that is recognizable as formalizing the type Self. The interaction of Self with object types is the subject of section 7.

We first introduce universal quantifiers and existential quantifiers. From existential quantifiers and recursion we define the Self quantifier. For the purpose of defining Self, only existentials and recursion are needed; one could dispense with universals. For the purposes of object-oriented languages, only the Self quantifier is needed; one could dispense with most of the second-order baggage. However, universal quantifiers are still useful in an object-oriented language to provide polymorphism, and existential quantifiers to provide data abstraction.

6.1 The Universal Quantifier

Polymorphism is obtained from the $\Delta_{\mathbf{X}}$ fragment for type variables of section 5.1, plus the following fragment for universal quantifiers. As before, we use the notation b{X} and B{X} to single out the occurrences of X free in b and in B; moreover, when X is clear from the context, b{A} stands for b{X←A}, and B{A} stands for B{X←A}, .

We write ∀(X)B{X} for the type of those polymorphic functions λ(X)b{X} that for any type A produce a result b{A} of type B{A}. Thus, b{A} is an instance of the polymorphic function for a specific type A; it can be obtained as the result of a type application (λ(X)b{X})(A).

$\Delta_{\forall}$

(Type All)	(Val Fun2)	(Val Appl2)
$\dfrac{E,X \vdash B}{E \vdash \forall(X)B}$	$\dfrac{E,X \vdash b : B}{E \vdash \lambda(X)b : \forall(X)B}$	$\dfrac{E \vdash b : \forall(X)B\{X\} \quad E \vdash A}{E \vdash b(A) : B\{A\}}$

The second-order λ-calculus $\mathbf{F}_{\mathbf{2}}$, usually called **F** [Girard, Lafont, Taylor 1989], is assembled below, along with related calculi. The built-in constant types are left out because free algebras can be encoded within **F** [Böhm, Berarducci 1985].

Starting in section 6.2, we include existential quantifiers in all our second-order calculi. We temporarily indicate calculi without existentials by italic letters:

$$\begin{aligned} \boldsymbol{F} &\triangleq \Delta_{\mathbf{x}} \cup \Delta_{\rightarrow} \cup \Delta_{\mathbf{X}} \cup \Delta_{\forall} & \qquad \boldsymbol{F}_{\mu} &\triangleq \boldsymbol{F} \cup \Delta_{\mu} \\ \boldsymbol{Ob} &\triangleq \Delta_{\mathbf{x}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{\mathbf{X}} \cup \Delta_{\forall} & \qquad \boldsymbol{Ob}_{\mu} &\triangleq \boldsymbol{Ob} \cup \Delta_{\mu} \\ \boldsymbol{FOb} &\triangleq \boldsymbol{F} \cup \Delta_{\mathbf{Ob}} & \qquad \boldsymbol{FOb}_{\mu} &\triangleq \boldsymbol{FOb} \cup \Delta_{\mu} \end{aligned}$$

The encoding of function types as object types extends trivially to second-order calculi, so ***Ob*** can encode ***F*** and ***FOb***. It is evident that ***F*** cannot encode ***Ob***, because ***F*** is strongly normalizing. However, $\boldsymbol{F}_{\mu}$ can encode ***Ob*** (and $\boldsymbol{Ob}_{\mu}$) using a typed version of the self-application semantics.

We move along directly to second-order calculi with subtyping. We write $\forall(X<:A)B\{X\}$ for the type of those bounded polymorphic functions $\lambda(X<:A)b\{X\}$ that for any subtype A' of A produce a result $b\{A'\}$ of type $B\{A'\}$.

We reuse the $\Delta_{<:\mathbf{X}}$ fragment of section 5.2, and add:

$\Delta_{<:\forall}$

(Type All<:)
$$\frac{E,X<:A \vdash B}{E \vdash \forall(X<:A)B}$$

(Sub All)
$$\frac{E \vdash A' <: A \qquad E,X<:A' \vdash B <: B'}{E \vdash \forall(X<:A)B <: \forall(X<:A')B'}$$

(Val Fun2<:)
$$\frac{E,X<:A \vdash b : B}{E \vdash \lambda(X<:A)b : \forall(X<:A)B}$$

(Val Appl2<:)
$$\frac{E \vdash b : \forall(X<:A)B\{X\} \qquad E \vdash A'<:A}{E \vdash b(A') : B\{A'\}}$$

$\Delta_{=<:\forall}$

(Eq Fun2<:)
$$\frac{E,X<:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X<:A)b \leftrightarrow \lambda(X<:A)b' : \forall(X<:A)B}$$

(Eq Appl2<:)
$$\frac{E \vdash b \leftrightarrow b' : \forall(X<:A)B\{X\} \qquad E \vdash A'<:A}{E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}}$$

(Eval Beta2<:)
$$\frac{E \vdash \lambda(X<:A)b : \forall(X<:A)B \qquad E \vdash C <: A}{E \vdash (\lambda(X<:A)b)(C) \leftrightarrow b\{X\leftarrow C\} : B\{X\leftarrow C\}}$$

(Eval Eta2<:)
$$\frac{E \vdash b : \forall(X<:A)B \qquad X \notin dom(E)}{E \vdash \lambda(X<:A)b(X) \leftrightarrow b : \forall(X<:A)B}$$

Thus, we obtain the calculi:

$$\boldsymbol{F}_{<:} \triangleq \Delta_{\mathbf{x}} \cup \Delta_{\rightarrow} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\rightarrow} \cup \Delta_{<:\forall} \qquad \boldsymbol{F}_{<:\mu} \triangleq \boldsymbol{F}_{<:} \cup \Delta_{<:\mu}$$
$$\boldsymbol{Ob}_{<:} \triangleq \Delta_{\mathbf{x}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mathbf{Ob}} \cup \Delta_{<:\forall} \qquad \boldsymbol{Ob}_{<:\mu} \triangleq \boldsymbol{Ob}_{<:} \cup \Delta_{<:\mu}$$
$$\boldsymbol{FOb}_{<:} \triangleq \boldsymbol{F}_{<:} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:\mathbf{Ob}} \qquad \boldsymbol{FOb}_{<:\mu} \triangleq \boldsymbol{FOb}_{<:} \cup \Delta_{<:\mu}$$

$\boldsymbol{F}_{<:}$ is described in [Cardelli, *et al.* 1991], but we assume only the simpler equational theory used in [Curien, Ghelli 1992]. The $\Delta_{<:\mu}$ fragment of $\boldsymbol{F}_{<:\mu}$ uses simpler rules for subtyping recursive types than the ones described in [Amadio, Cardelli 1991].

While adding subtyping and polymorphism to our calculi, we have consciously chosen not to modify the $\Delta_{\mathbf{Ob}}$ fragment. However, it is tempting to change the (Val Override) rule as follows:

(Val Structural Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)
$$\frac{E \vdash C <: A \qquad E \vdash a : C \qquad E, x:C \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:C)b : C}$$

The difference between (Val Override) and (Val Structural Override) can be seen when C is a type variable, as in the following example:

$\lambda(C<:[l:Nat])\lambda(a:C)a.l:=3 : \forall(C<:[l:Nat])C\rightarrow[l:Nat]$ with (Val Override)

$\lambda(C<:[l:Nat])\lambda(a:C)a.l:=3 : \forall(C<:[l:Nat])C\rightarrow C$ with (Val Structural Override)

This new rule appears intuitively sound. It implicitly relies on the invariance of object types, and on the "structural subtyping" assumption that every subtype of an object type is an object type. Such an assumption is quite easily realized in programming languages, and can be shown as a theorem in formal systems such as ours.

However, structural subtyping is not satisfied by the semantics of section 9, where the subtyping relation is simply the "unstructured" subset relation. In that semantics, the type $\forall(C<:[l:Nat])C\rightarrow C$ contains only an identity function and its approximations, and hence does not contain the above program. This does not seem to be a special property of our semantics, but rather a consequence of the parametricity of polymorphic functions [Reynolds 1983].

A semantics that satisfies structural subtyping could probably be devised, but would be more complex and might have a rather syntactic flavor (see [Cardelli, Mitchell 1991] for a first-order example of such a semantics). Hence we do not adopt the rule (Val Structural Override), but we return to the topic of structural assumptions later on, in section 8.3.

6.2 The Existential Quantifier

An existentially quantified type $\exists(X<:A)B\{X\}$ is the type of the pairs <A',b> where A' is a subtype of A and b is a term of type B{A'}. The type $\exists(X<:A)B\{X\}$ can be seen as a (partially) abstract data type with interface B{X} and with representation type X known only to be a subtype of A. A pair <A',b> describes an element of the abstract data type with representation type A' and implementation b{A'} [Cardelli, Wegner 1985; Mitchell, Plotkin 1985]. In order to be fully explicit, we write the pair <A',b> more verbosely in the form:

pack X<:A=A', b{X}:B{X}

A pair c of type $\exists(X<:A)B\{X\}$ can be used in the construct:

open c as X<:A,x:B{X} in d{X,x}:D

where d has access to the representation type X and implementation x, and must produce a result of a type D that does not depend on X. The requirement that X does not occur in D is necessary in order to preserve the abstraction. At evaluation time, if c is <A',b>, then the result is d{A',b}.

Within $\boldsymbol{F}_{<:}$ (and hence $\boldsymbol{FOb}_{<:}$) it is possible to encode bounded existential quantifiers by setting:

$\exists(X<:A)B\{X\} \triangleq \forall(Y<:Top)(\forall(X<:A)B\{X\}\rightarrow Y)\rightarrow Y$ (Y not occurring in A or B)

with the following definitions:

pack X<:A=C, b{X}:B{X} $\triangleq$ $\lambda(Y<:Top)\lambda(f:\forall(X<:A)B\{X\}\rightarrow Y)f(C)(b\{C\})$

open c as X<:A,x:B{X} in d{X,x}:D $\triangleq$ $c(D)(\lambda(X<:A)\lambda(x:B\{X\})d\{X,x\})$

The encoding of existentials, however, does not satisfy a natural η rule. Hence, we take existential as primitive, with the rules given below. All these rules are derivable from the encoding, except for the η rule (Eq Repack<:).

$\Delta_{<:\exists}$

(Type Exists<:)

$$\frac{E,X<:A \vdash B}{E \vdash \exists(X<:A)B}$$

(Sub Exists)

$$\frac{E \vdash A <: A' \quad E,X<:A \vdash B <: B'}{E \vdash \exists(X<:A)B <: \exists(X<:A')B'}$$

(Val Pack<:)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash (\text{pack } X<:A=C,\ b\{X\}:B\{X\}) : \exists(X<:A)B\{X\}}$$

(Val Open<:)

$$\frac{E \vdash c : \exists(X<:A)B \quad E \vdash D \quad E,X<:A,x:B \vdash d : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B \text{ in } d:D) : D}$$

For future reference, note that existentials are covariant in their bounds. Moreover, if B is covariant in X (written $B\{X^+\}$), then $\exists(X<:A)B\{X^+\}$ is isomorphic to $B\{A^+\}$. This isomorphism does not hold otherwise; for example, $\exists(X<:A)[l_i:B_i\{X^+\}^{\,i\in 1..n}]$ is not isomorphic to $[l_i:B_i\{A^+\}^{\,i\in 1..n}]$ since $[l_i:B_i\{X^+\}^{\,i\in 1..n}]$ is not covariant in X, even though each $B_i\{X^+\}$ is. This observation will play an important role in section 7.

$\Delta_{=<:\exists}$

(Eq Pack<:)

$$\frac{E \vdash C <: A' \quad E \vdash A'<:A \quad E,X<:A' \vdash B'\{X\}<:B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash (\text{pack } X<:A=C,\ b\{X\}:B\{X\}) \ \leftrightarrow\ (\text{pack } X<:A'=C,\ b'\{X\}:B'\{X\}) : \exists(X<:A)B\{X\}}$$

(Eq Open<:)

$$\frac{E \vdash c \leftrightarrow c' : \exists(X<:A)B \quad E \vdash D \quad E,X<:A,x:B \vdash d \leftrightarrow d' : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B \text{ in } d:D) \ \leftrightarrow\ (\text{open } c' \text{ as } X<:A,x:B \text{ in } d':D) : D}$$

(Eval Unpack<:) (where $c \equiv \text{pack } X<:A=C,\ b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X<:A)B\{X\} \quad E \vdash D \quad E,X<:A,x:B\{X\} \vdash d\{X,x\} : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B\{X\} \text{ in } d\{X,x\}:D) \ \leftrightarrow\ d\{C,b\{C\}\} : D}$$

(Eval Repack<:)

$$\frac{E \vdash b : \exists(X<:A)B\{X\} \quad E,y:\exists(X<:A)B\{X\} \vdash d\{y\} : D}{E \vdash (\text{open } b \text{ as } X<:A,x:B\{X\} \text{ in } d\{\text{pack } X'<:A=X,\ x:B\{X'\}\}:D) \ \leftrightarrow\ d\{b\} : D}$$

The rules (Eq Pack<:) and (Eq Open<:) are congruence rules. The encoding of (Eq Pack<:) is derivable in $\boldsymbol{F}_{<:}$ using the Domain Restriction Lemma ([Cardelli, *et al.* 1991], section 2.4). This rule takes subtyping into account, equating pack term with different type bounds A and A' and type bodies $B\{X\}$ and $B'\{X\}$. A more flexible congruence rule is derivable from (Eq Pack<:):

(Eq Pack<: Lemma)

$$\frac{E \vdash C <: A' \qquad E \vdash A' <: A \qquad E, X<:A' \vdash B'\{X\} <: B\{X\} \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : B\{C\} \qquad E \vdash b'\{C\} : B'\{C\}}{E \vdash (\text{pack } X<:A=C,\ b\{X\}:B\{X\}) \ \leftrightarrow\ (\text{pack } X<:A'=C,\ b'\{X\}:B'\{X\}) : \exists(X<:A)B\{X\}}$$

For the derivation of (Eq Pack<: Lemma), we use (Eq Pack<:) twice, first with b≡b' and then with A≡A' and B≡B', and apply transitivity to the two results.

The rules (Eval Unpack<:) and (Eval Repack<:) are evaluation rules. The rule (Eval Unpack<:) requires c to contain the types A and B mentioned in the open construct. This may not be the case if c has been obtained via subsumption: c could have the form pack X<:A'=C, b:B' where A'<:A and B'<:B. However, then, the rules (Eq Pack<:) and (Eq Open<:) can be used to replace the inadequate c with an adequate one.

The equational rules are somewhat conservative. In particular, they do not equate pack terms based on different representation types. For example, we might expect that the terms (pack X<:Int=Nat, $(0,\text{succ}_{\text{Nat}})$) and (pack X<:Int=Int, $(0,\text{succ}_{\text{Int}})$) to be equal at type $\exists(X<:A)X\times(X\to X)$, but this cannot be proved. This can be proved using parametricity in the theory of [Plotkin, Abadi, Cardelli 1993]. For simplicity we restrict attention to our conservative rules, although more ambitious rules may have advantages in combination with objects.

We now add existential quantifiers to our calculi. For calculi with $\forall$ and $\to$, the only contribution of the $\Delta_\exists$ fragment is the rule (Eq Repack<:), because of the encoding of $\exists$ in terms of $\forall$. For calculi without $\to$ ($\mathbf{Ob}_{<:}$ and $\mathbf{Ob}_{<:\mu}$), which can encode only invariant function types, the $\Delta_\exists$ fragment is needed also to validate the rule (Sub Exists).

$$\begin{array}{rcl@{\qquad}rcl}
\mathbf{F} & \triangleq & \boldsymbol{F} \cup \Delta_\exists & \mathbf{F}_\mu & \triangleq & \boldsymbol{F}_\mu \cup \Delta_\exists \\
\mathbf{Ob} & \triangleq & \boldsymbol{Ob} \cup \Delta_\exists & \mathbf{Ob}_\mu & \triangleq & \boldsymbol{Ob}_\mu \cup \Delta_\exists \\
\mathbf{FOb} & \triangleq & \boldsymbol{FOb} \cup \Delta_\exists & \mathbf{FOb}_\mu & \triangleq & \boldsymbol{FOb}_\mu \cup \Delta_\exists \\
\mathbf{F}_{<:} & \triangleq & \boldsymbol{F}_{<:} \cup \Delta_{<:\exists} & \mathbf{F}_{<:\mu} & \triangleq & \boldsymbol{F}_{<:\mu} \cup \Delta_{<:\exists} \\
\mathbf{Ob}_{<:} & \triangleq & \boldsymbol{Ob}_{<:} \cup \Delta_{<:\exists} & \mathbf{Ob}_{<:\mu} & \triangleq & \boldsymbol{Ob}_{<:\mu} \cup \Delta_{<:\mu} \\
\mathbf{FOb}_{<:} & \triangleq & \boldsymbol{FOb}_{<:} \cup \Delta_{<:\exists} & \mathbf{FOb}_{<:\mu} & \triangleq & \boldsymbol{FOb}_{<:\mu} \cup \Delta_{<:\exists}
\end{array}$$

In the next section we show that $\mathbf{Ob}_{<:}$ can encode $\mathbf{F}_{<:}$, and that $\mathbf{Ob}_{<:\mu}$ can encode $\mathbf{F}_{<:\mu}$. It is unknown whether $\mathbf{Ob}_{<:}$ can be encoded in, say, $\mathbf{F}_{<:\mu}$. We leave as an open problem whether $\mathbf{Ob}_{<:}$ can be encoded in any of the standard typed λ-calculi while preserving subtyping; section 10 deals with encodings that translate subtypings into coercions.

6.3 An Encoding of Variant Product and Function Types

Within first-order object calculi we were unable to find a full translation of λ-calculi with subtyping, because the variance properties of $\to$ could not be obtained with invariant object types only (see section 4.1). Now, however, we have at our disposal bounded universal types, which are contravariant in their bounds, and bounded existential types, which are covariant in their bounds. Combining these variant constructions with invariant object types, we can define a variant version of function types:

$$A \mathbin{\dot{\to}} B \ \triangleq\ \forall(X<:A)\,\exists(Y<:B)\,[\text{arg}:X,\ \text{val}:Y]$$

We obtain $A \mathbin{\dot{\to}} B <: A' \mathbin{\dot{\to}} B'$ if $A' <: A$ and $B <: B'$.

This idea gives rise to an encoding of $\mathbf{F}_{\mathbf{1}<:}$ into $\mathbf{Ob}_{<:}$. As before, this encoding is defined on type derivations, but for simplicity we write it as a translation of type-annotated λ-terms.

Translation of the first-order λ-calculus with subtyping

$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$

$\langle\!\langle E,x:A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x:\langle\!\langle A \rangle\!\rangle$

$\langle\!\langle K \rangle\!\rangle \triangleq K$

$\langle\!\langle A \to B \rangle\!\rangle \triangleq$

$\quad \forall(X<:\langle\!\langle A \rangle\!\rangle)\ \exists(Y<:\langle\!\langle B \rangle\!\rangle)\ [arg:X, val:Y] \qquad \rho \in Var \to \mathbf{Ob}_{<:}\text{-term}$

$\langle\!\langle x_A \rangle\!\rangle_\rho \triangleq \rho(x) \qquad \emptyset(x) \triangleq x$

$\langle\!\langle k_K \rangle\!\rangle_\rho \triangleq k \qquad (\rho\{y \leftarrow a\})(x) \triangleq$ if x=y then a else $\rho(x)$

$\langle\!\langle b_{A\to B}(a_A) \rangle\!\rangle_\rho \triangleq$

$\quad$ open $\langle\!\langle b \rangle\!\rangle_\rho(\langle\!\langle A \rangle\!\rangle)$ as $Y<:\langle\!\langle B \rangle\!\rangle$, $y:[arg:\langle\!\langle A \rangle\!\rangle, val:Y]$

$\quad$ in $(y.arg \Leftarrow \varsigma(x:[arg:\langle\!\langle A \rangle\!\rangle, val:Y])\ \langle\!\langle a \rangle\!\rangle_\rho).val \qquad$ for $Y,y,x \notin FV(\langle\!\langle a \rangle\!\rangle_\rho)$

$\langle\!\langle \lambda(x:A)b_B \rangle\!\rangle_\rho \triangleq$

$\quad \lambda(X<:\langle\!\langle A \rangle\!\rangle)$

$\qquad$ (pack $Y<:\langle\!\langle B \rangle\!\rangle = \langle\!\langle B \rangle\!\rangle$,

$\qquad\quad$ [arg = $\varsigma(x:[arg:X, val:\langle\!\langle B \rangle\!\rangle])\ x.arg$,

$\qquad\quad$ val = $\varsigma(x:[arg:X, val:\langle\!\langle B \rangle\!\rangle])\ \langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow x.arg\}}$]

$\qquad$: $[arg:X, val:Y]$)

This translation can be extended to recursive types since $A \dot{\to} B$ is contractive in any X, and can also be extended trivially to second-order quantifiers. Hence our largest calculus, $\mathbf{FOb}_{<:\mu}$, can be embedded inside $\mathbf{Ob}_{<:\mu}$. We therefore consider $\mathbf{Ob}_{<:\mu}$ as our final pure object calculus.

Trivially, we can obtain covariant product types, since these can be represented in terms of universal quantifiers and variant function types in $\mathbf{F}_{<:}$. However, a more direct encoding is available:

$$A \dot{\times} B \triangleq \exists(X<:A)\ \exists(Y<:B)\ [fst:X, snd:Y]$$

We obtain $A \dot{\times} B <: A' \dot{\times} B'$ if $A <: A'$ and $B <: B'$.

In [Cardelli 1991] it is shown that record types $\langle l_i:A_i\ ^{i\in 1..n}\rangle$ can be represented in $\mathbf{F}_{<:}$, using covariant product types. These record types are covariant, in the sense that $\langle l_i:A_i\ ^{i\in 1..n+m}\rangle <: \langle l_i:A'_i\ ^{i\in 1..n}\rangle$ if $A_i <: A'_i$ for $i \in i..n$. Trivially, then, we can encode covariant record types in $\mathbf{Ob}_{<:}$.

6.4 An Encoding of Variant Object Types

In section 4.5 we observed that a general covariance rules for objects is unsound. However, as in the encoding of the λ-calculus just given, we can make covariant any component whose method is only invoked (like val), and contravariant any component whose method is only overridden (like arg).

The idea for obtaining covariance is exactly the one used for the λ-calculus. We rewrite an object type [m:B, ...] as $\exists(Y<:B)\ [m:Y, ...]$. The former is invariant in B, while the latter is covariant in B. The existential quantifier still allows the invocation of m, but blocks overrides of m from the outside, since the quantifier hides the representation type Y.

The idea for obtaining contravariance is more complicated; the technique used for the λ-calculus does not seem to generalize. To make a component m:B contravariant in an object type $A \triangleq [m:B, ...]$, we first rewrite the type as $A' \triangleq \mu(X)\ \exists(Y<:(X\to B)\to X)\ [m_{in}:Y, m:B, ...]$, where m_{in} is a new method

name. Note that A' is still invariant in B. We simulate an override to the method m of an object o by writing the invocation $o.m_{in}(\lambda(s:A')b')$ instead of $o.m \Leftarrow \varsigma(s:A)b$, where b' imitates b. Our intent is that an invocation of m_{in} with argument $\lambda(s:A')b'$ will override m internally. Hence the code for a typical object o of type A' will be:

$$in([m_{in} = \varsigma(s:UA')\ \lambda(f: A' \to B)\ in(s.m \Leftarrow \varsigma(s:UA')\ f(in(s))),\ m = \varsigma(s:UA')\ s.m,\ ...\ ]): A'$$

where $UA' \triangleq [m_{in}:(A' \to B) \to A',\ m:B,\ ...]$, and for any a:UA'

$$in(a): A' \triangleq \text{fold}(A', \text{pack } Y<:(A' \to B) \to A' = (A' \to B) \to A', a : [m_{in}:Y,\ m:B,\ ...])$$

Finally we use subsumption to forget the m component, so that o has the type:

$$\mu(X)\ \exists(Y<:(X \to B) \to X)\ [m_{in}:Y,\ ...]$$

which is contravariant in B. Once o has this type, its method m cannot be invoked from the outside since it is not even visible.

These techniques for obtaining covariance and contravariance are suggestive but not fully satisfactory. For example, after making two components covariant, we are no longer able to reorder them, since $\exists(X<:A)\ \exists(Y<:B)\ C$ and $\exists(Y<:B)\ \exists(X<:A)\ C$ are not equivalent types. Still, these techniques may inspire an encoding or a semantics for a language with built-in object types with covariant and contravariant components.

6.5 The Self Quantifier

Within the second-order ς-calculus with bounded quantifiers and recursion, $\mathbf{Ob}_{<:\mu}$, we can encode an interesting construction that we call the *Self quantifier*. The Self quantifier is a combination of recursion and bounded existentials, with recursion going "through the bound":

$$\varsigma(X)B \triangleq \mu(Y)\exists(X<:Y)B \qquad (Y \text{ not occurring in } B)$$

In general, any type $B\{A\}$ can be transformed into a type $\exists(Y<:A)B\{Y\}$ covariant in A. (Recall though that these types are not always isomorphic.) An analogous technique applies to recursive types, and motivates our definition of the Self quantifier. Given an equation $X = B\{X\}$, we transform it into $X = \exists(Y<:X)B\{Y\}$. The solution to this equation, namely $\varsigma(X)B\{X\}$, satisfies the subtyping property:

$$\text{if } B\{Y\}<:B'\{Y\} \text{ then } \varsigma(X)B\{Y\} <: \varsigma(X)B'\{Y\},$$

even though we may not have $\mu(X)B\{X\} <: \mu(X)B'\{X\}$.

Modulo an unfolding, $\varsigma(X)B$ is the same as $\exists(X<:\varsigma(X)B)B$. Hence, by analogy with the interpretation of existential types, $\varsigma(X)B\{X\}$ can be understood informally as the type of pairs $\langle C, c\rangle$ consisting of a subtype C of $\varsigma(X)B\{X\}$ and an element c of $B\{C\}$.

For example, suppose we have an element x of type $\varsigma(X)X$. Then, choosing $\varsigma(X)X$ as the required subtype of $\varsigma(X)X$, we obtain $\langle \varsigma(X)X, x\rangle : \varsigma(X)X$. Therefore we can construct:

$$\mu(x)\ \langle \varsigma(X)X, x\rangle\ :\ \varsigma(X)X$$

Less trivially, suppose we want to build a memory cell m:M with a read operation rd:Nat and a write operation wr:Nat→M. We can define:

$$M \triangleq \varsigma(X)[rd:Nat,\ wr:Nat \to X]$$

where the wr method should use its argument to override the rd field. For convenience, we adopt the following abbreviation to unfold a Self quantifier:

Notation

$A(C) \triangleq B\{C\}$ whenever $A \equiv \varsigma(X)B\{X\}$ and $C<:A$

So, for example, $M(M) \equiv [rd:Nat, wr:Nat \to M]$.

To define a memory cell, we are going to use twice the fact that if $x:M(M)$, then $\langle M, x\rangle:M$ (simply by the typing rule for $\langle -,-\rangle$ explained previously). First, we need a method body for wr that, with self $s:M(M)$ and argument $n:Nat$, produces a result of type M. Since $s.rd:=n$ has the same type as s, namely $M(M)$, we can use $\langle M, s.rd:=n\rangle:M$ as the body of the wr method. Using this method, we can then build an object of type $M(M)$. Therefore, we have:

$$m: M \triangleq \langle M, [rd = 0, wr = \varsigma(s:M(M))\, \lambda(n:Nat)\, \langle M, s.rd:=n\rangle]\rangle$$

Beyond these initial and rather informal examples, the best way to understand the Self quantifier is to look at its specific application to objects in section 7. Still, it is important to examine the intrinsic properties of $\varsigma(X)B$, since they help in deriving some complex rules for object types with Self. The rest of this section is dedicated to the study of the abstract properties of types of the form $\varsigma(X)B$.

The two basic operations for ς are similar to the ones for existentials. One operation constructs an element of $\varsigma(X)B$, given a subtype of $\varsigma(X)B$ and an appropriate value; it is the composition of pack for existentials and fold for recursive types. The other operation inspects an element of $\varsigma(X)B$ (as much as possible) and computes with its contents; it is the composition of unfold for recursive types and open for existentials.

The operation for constructing elements of type $\varsigma(X)B$, in full generality, binds a type variable. Hence, we need a more complex syntax than the pairing $\langle -,-\rangle$ used above. We reuse the symbol ς for this second-order construct, in the same way we use λ for both first-order and second-order binders. We refine the notation $\langle C,b\rangle$ to $\varsigma(Y<:A=C)b$. The term $\varsigma(Y<:A=C)b$ binds C to Y in b, and requires C to be a subtype of A. Within b, the types Y and C are equivalent. The type of the whole term is A.

We define, for $A \equiv \varsigma(X)B\{X\}$, $C<:A$, and $b\{C\}:B\{C\}$:

$$\varsigma(Y<:A=C)\, b\{Y\} \triangleq \mathrm{fold}(A,\ (\mathrm{pack}\ Y<:A=C,\ b\{Y\}:B\{Y\}))$$

and, for $c:A$ and $d\{Y,y\}:D$, where Y does not occur in D:

$$(\mathrm{use}\ c\ \mathrm{as}\ Y<:A,\ y:B\{Y\}\ \mathrm{in}\ d\{Y,y\}:D) \triangleq (\mathrm{open}\ \mathrm{unfold}(c)\ \mathrm{as}\ Y<:A,\ y:B\{Y\}\ \mathrm{in}\ d\{Y,y\}:D)$$

Now, the following rules for the Self quantifier can be derived from the rules for μ and $\exists$:

Δ_ς

(Type Self)

$$\frac{E,X \vdash B \quad B \succ X}{E \vdash \varsigma(X)B}$$

(Sub Self)

$$\frac{E,X \vdash B <: B' \quad B,B' \succ X}{E \vdash \varsigma(X)B <: \varsigma(X)B'}$$

(Val Self) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} : A}$$

(Val Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c : A \quad E \vdash D \quad E, Y<:A, y:B\{Y\} \vdash d : D}{E \vdash (\text{use } c \text{ as } Y<:A, y:B\{Y\} \text{ in } d:D) : D}$$

$\Delta_{=\varsigma}$

(Eq Self) (where $A \equiv \varsigma(X)B\{X\}$, $A' \equiv \varsigma(X)B'\{X\}$)

$$\frac{E \vdash C <: A' \quad E \vdash A \quad E,X \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} \leftrightarrow \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c \leftrightarrow c' : A \quad E \vdash D \quad E, Y<:A, y:B\{Y\} \vdash d \leftrightarrow d' : D}{E \vdash (\text{use } c \text{ as } Y<:A, y:B\{Y\} \text{ in } d:D) \leftrightarrow (\text{use } c' \text{ as } Y<:A, y:B\{Y\} \text{ in } d':D) : D}$$

(Eval Unself) (where $A \equiv \varsigma(X)B\{X\}$, $c \equiv \varsigma(Z<:A=C)b\{Z\}$)

$$\frac{E \vdash c : A \quad E \vdash D \quad E,Y<:A,y:B\{Y\} \vdash d\{Y,y\} : D}{E \vdash (\text{use } c \text{ as } Y<:A,y:B\{Y\} \text{ in } d\{Y,y\}:D) \leftrightarrow d\{C,b\{C\}\} : D}$$

(Eval Reself) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash b : A \quad E,y:A \vdash d\{y\} : D}{E \vdash (\text{use } b \text{ as } Y<:A,y:B\{Y\} \text{ in } d\{\varsigma(Y'<:A=Y)y\} :D) \leftrightarrow d\{b\} : D}$$

Notation We write:

- $\varsigma\langle A,c\rangle$ for $\varsigma(X<:A=A)c$ when X does not occur in c
- $\varsigma(X=A)c\{X\}$ for $\varsigma(X<:A=A)c\{X\}$

Note the desired rule of covariance, (Sub Self). This rule holds because existential types are covariant in their bounds, and because recursion in $\mu(Y)\exists(X<:Y)B\{X\}$ involves only a single covariant position. It can be verified as follows:

$E,X \vdash B <: B'$
$\Rightarrow E,Z,Y<:Z,X<:Y \vdash B <: B'$ by standard weakening lemmas, for fresh Y,Z
$\Rightarrow E,Z,Y<:Z \vdash \exists(X<:Y)B <: \exists(X<:Z)B'$ by (Sub Exists)
$\Rightarrow E \vdash \mu(Y)\exists(X<:Y)B <: \mu(Z)\exists(X<:Z)B'$ by (Sub Rec) and contractiveness rules

Much as we have done for the (Eq Pack<:) rule for existentials, we derive a flexible congruence rule from (Eq Self):

(Eq Self Lemma) (where $A \equiv \varsigma(X)B\{X\}$, $A' \equiv \varsigma(X)B'\{X\}$)

$$\frac{E \vdash C <: A' \quad E \vdash A \quad E,X \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B\{C\} \quad E \vdash b'\{C\} : B'\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} \leftrightarrow \varsigma(Y<:A'=C)b'\{Y\} : A}$$

The memory cell definition can now be understood formally, with the $\varsigma\langle M,...\rangle$ notation just introduced replacing the informal notation $\langle M,...\rangle$. We rewrite the example explicitly, taking advantage of the new ς-binders:

$M \triangleq \varsigma(\text{Self})[\text{rd:Nat, wr:Nat}\to\text{Self}]$

$m{:}\,M \triangleq \varsigma(\text{Self=M})\,[\text{rd} = 0, \text{wr} = \varsigma(\text{s:M(Self)})\,\lambda(\text{n:Nat})\,\varsigma\langle\text{Self, s.rd:=n}\rangle]\rangle$

Later examples frequently adopt this choice of Self as a bound variable.

6.6 A Pure Second-Order Object Calculus

We conclude this section by considering a pure second-order object calculus, $\varsigma\mathbf{Ob}$, based exclusively on object types and the Self quantifier. (See appendix D.)

$$\varsigma\mathbf{Ob} \triangleq \Delta_{\mathbf{X}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mathbf{Ob}} \cup \Delta_{\varsigma}$$

This calculus departs from second-order λ-calculi by omitting function types and the standard quantifiers. It seems that, in $\varsigma\mathbf{Ob}$, the only function types that can be encoded are invariant, and that the standard second-order quantifiers are not expressible. Still, as the next section demonstrates, the Self quantifier provides an essential second-order feature of object calculi, namely the type Self, with a form of type recursion. Interestingly, then, $\varsigma\mathbf{Ob}$ is a very small second-order object calculus that covers a spectrum of object-oriented notions.

7. Object Types with Self

The payoff of the Self quantifier, and particularly of the (Sub Self) rule derived in section 6.5, comes when it is used in conjunction with object types. Object types with Self are obtained by the combination of the simple object types of sections 3 and 4 with the Self quantifier of section 6.5. These new types allow subsumption between objects containing methods that return self.

In this section, we first derive rules for object types with Self. We then show how, using these derived rules, we can easily provide typings for the four examples of section 2.4. We work entirely within $\mathbf{Ob}_{<:\mu}$, that is, within the second-order calculus with bounded quantifiers, recursion, and simple object types. Throughout, we omit some of the routine rule derivations, and include only a few interesting ones.

7.1 ς-Objects

In this section we examine types of the form:

$\varsigma(X)[l_i{:}B_i\{X^+\}^{\,i\in 1..n}]$ where $B\{X^+\}$ indicates that X occurs only covariantly in B

We call these structures ς-object types, and ς-objects their corresponding values. The parameter X in $\varsigma(X)[l_i{:}B_i\{X^+\}^{\,i\in 1..n}]$ is intended as the Self type hypothesized in section 5.5. Note that we do not require that $[l_i{:}B_i\{X\}^{\,i\in 1..n}]$ be covariant in X, only that each $B_i\{X\}$ be covariant in X.

Although ς-object types are obtained by applying a Self quantifier (which has no covariance restrictions) to an object type, for the most part we consider $\varsigma(X)[l_i{:}B_i\{X^+\}^{\,i\in 1..n}]$ as a single type construction. We shall see shortly that the covariance requirement is necessary when selecting components of ς-objects. For emphasis, we use a special syntax for the combination of the Self quantifier, object types, and the covariance requirement:

Notation

$\varsigma(X^+)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \triangleq \varsigma(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}]$ when X occurs only covariantly in the B_i

The covariance requirement implies that X_i must not occur within any object type within B_i, since object types are invariant in their components. For example, the type $\varsigma(X)[l{:}\ \varsigma(Y)[m{:}X, n{:}Y]]$ violates

the covariance requirement. Hence, informally, we may say that Self types do not nest: there is a single meaningful Self type within each pair of object brackets.

It is often useful to consider an unfolding A(C) of a ς-object type A:

Notation

$A(C) \triangleq [l_i:B_i\{C\}^{\,i\in 1..n}]$ whenever $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and $C<:A$

We frequently consider A(X), for a variable X, and the *self-unfolding* A(A) of A. (When building an element of type A it is common to build first an element of type A(A).) We say that a type C<:A and an element of A(C) constitute an *implementation* of A, since they can be used to build an element of A. Then C is the *representation type* for the implementation.

The operations on ς-objects are defined as follows. Assume a:A with $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and $A(X)\equiv[l_i:B_i\{X^+\}^{\,i\in 1..n}]$, and set, with some overloading of notation:

$a.l_j \triangleq$
(use a as Z<:A, y:A(Z) in $y.l_j : B_i\{A^+\}$)

$a.l_j \Leftarrow \varsigma(Y<:A, y:A(Y), x:A(Y))b\{Y,y,x\} \triangleq$
(use a as Z<:A, y:A(Z) in $\varsigma(Y<:A=Z)\ (y.l_j \Leftarrow \varsigma(x:A(Y))b\{Y,y,x\}) : A$)

Then, combining the rules for objects and for Self quantification we derive the following rules.

$\Delta_{\varsigma+}$

(Type ςObject)

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i\in 1..n \qquad (l_i \text{ distinct})}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Sub ςObject)

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i\in 1..n+m \qquad (l_i \text{ distinct})}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Val ςObject) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash C<:A \quad E \vdash b\{C\} : A(C)}{E \vdash \varsigma(Y<:A=C)\ b\{Y\} : A}$$

(Val ςSelect) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \quad j\in 1..n}{E \vdash a.l_j : B_j\{A^+\}}$$

(Val ςOverride) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \quad E, Y<:A, y:A(Y), x:A(Y) \vdash b : B_j\{Y^+\} \quad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, y:A(Y), x:A(Y))b : A}$$

$\Delta_{=\varsigma+}$

(Eq ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

$$\frac{E \vdash C <: A' \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : A'(C)}{E \vdash \varsigma(Y<:A=C)b\{Y\} \ \leftrightarrow \ \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Sub ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

$$\frac{E \vdash C <: A' \qquad E, x_i:A(C) \vdash b_i\{C\} : B_i\{C\} \ \ \forall i\in 1..n \qquad E, x_j:A'(C) \vdash b_j\{C\} : B_j\{C\} \ \ \forall j\in n+1..m}{E \vdash \varsigma(Y<:A=C)[l_i=\varsigma(x_i:A(Y))b_i\{Y\}^{\ i\in 1..n}] \ \leftrightarrow \ \varsigma(Y<:A'=C)[l_i=\varsigma(x_i:A'(Y))b_i\{Y\}^{\ i\in 1..n+m}] : A}$$

(Eq ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \qquad j\in 1..n}{E \vdash a.l_j \ \leftrightarrow \ a'.l_j : B_j\{A^+\}}$$

(Eq ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \qquad E, Y<:A, y:A(Y), x:A(Y) \vdash b \leftrightarrow b' : B_j\{Y^+\} \qquad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, y:A(Y), x:A(Y))b \ \leftrightarrow \ a'.l_j \Leftarrow \varsigma(Y<:A, y:A(Y), x:A(Y))b' : A}$$

(Eval ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \qquad j\in 1..n}{E \vdash c.l_j \ \leftrightarrow \ a\{C\}.l_j : B_j\{A^+\}}$$

(Eval ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \qquad E, Y<:A, y:A(Y), x:A(Y) \vdash b\{Y,y,x\} : B_j\{Y^+\} \qquad j\in 1..n}{E \vdash c.l_j \Leftarrow \varsigma(Y<:A,y:A(Y),x:A(Y))b\{Y,y,x\} \ \leftrightarrow \ \varsigma(Z<:A=C)\ a\{Z\}.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,a\{Z\},x\} : A}$$

The most remarkable fact is that the (Sub ςObject) rule holds for ς-object types. We recall that at the end of section 5 we found that the analogous rule for recursive object types did not hold.

The (Val ςObject) rule can be used to build a ς-object $\varsigma(Y<:A=C)b\{Y\}$ from a subtype C of the desired ς-object type A, and from a regular object $b\{C\}$. The Y variable in $b\{Y\}$ is the Self type, in case the methods of b need to refer to it.

When building a ς-object by (Val ςObject) we can take C<:A to be a concrete object type, often choosing C=A; we rarely need to work parametrically for an arbitrary X<:A. However, the generality of using an arbitrary subtype of A is critical in a rule for override shown later. In section 8.1 we will see that this generality has a price.

The (Eq ςObject) rule yields a flexible congruence lemma, analogous to the one obtained from the (Eq Pack<:) rule for existentials:

(Eq ςObject Lemma) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

$$\frac{E \vdash C <: A' \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : A(C) \qquad E \vdash b'\{C\} : A'(C)}{E \vdash \varsigma(Y<:A=C)b\{Y\} \ \leftrightarrow \ \varsigma(Y<:A'=C)b'\{Y\} : A}$$

The (Eq Sub ςObject) rule is obtained from (Eq Sub Object) and (Eq Self Lemma). It is of limited power because the same C appears on both sides of the conclusion. We can trace back this limitation to a similar one in the rules for existential types.

The ς-object selection operation reduces fairly simply to a regular selection operation on the underlying object.

The ς-object override operation is more interesting, although similarly it reduces to an override operation on the underlying object. The overriding method b can take advantage of three variables: (1) Y<:A, the unknown subtype of A that was used to construct a; (2) y:A(Y), the raw object inside a, which can be thought of as the *old self*; y is the value of self at the time the override takes place, containing the old version of method l_j; (3) x:A(Y), the regular self of the overriding method b. From these variables, b must produce a result of type $B_j\{Y^+\}$, parametrically in Y.

We now verify some of these rules in detail.

- (Val ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

  The derivation relies on covariance of B_i in X, and subsumption.

  E, Y<:A, y:A(Y) ⊢ $y.l_j : B_j\{Y^+\}$ by (Val Select)

  E, Y<:A, y:A(Y) ⊢ $B_j\{Y^+\} <: B_j\{A^+\}$ since Y<:A and $B_j\{Y^+\}$ is covariant in Y

  E, Y<:A, y:A(Y) ⊢ $y.l_j : B_j\{A^+\}$ by (Val Subsumption)

  E ⊢ (use a as Y<:A, y:A(Y) in $y.l_j : B_j\{A^+\}$) : $B_j\{A^+\}$ by (Val Use) since E ⊢ a:A

- (Val ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

  The derivation relies on the full power of (Val Self). Note that y:A(Y) does not imply y:A(A), even though Y<:A. Otherwise b would only need to have type $B_i\{A^+\}$, and this would permit unsound overrides.

  E, Y<:A, y:A(Y), x:A(Y) ⊢ $b\{Y,y,x\} : B_j\{Y^+\}$ by assumption

  E, Y<:A, y:A(Y) ⊢ $y.l_j \Leftarrow \varsigma(x:A(Y))b\{Y,y,x\} : A(Y)$ by (Val Override)

  E, Y<:A, y:A(Y) ⊢ $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,y,x\} : A$ by (Val Self)

  E ⊢ (use a as Y<:A, y:A(Y) in $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,y,x\}$:A) : A by (Val Use)

- (Eq ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

  E ⊢ A' <: A by (Sub Self)

  E ⊢ $b\{C\} \leftrightarrow b'\{C\} : A'(C)$ by assumption

  E ⊢ (pack Y<:A=C, $b\{Y\}:A(Y)$) ↔

  (pack Y<:A'=C, $b'\{Y\}:A'(Y)$) : $\exists(X<:A)A(X)$ by (Eq Pack<:)

  E ⊢ $b\{C\} \leftrightarrow b'\{C\} : A'(C)$ by assumption

  E ⊢ $b'\{C\} : A'(C)$ by a standard lemma

  E ⊢ (pack Y<:A'=C, $b'\{Y\}:A'(Y)$) : $\exists(X<:A')A'(X)$ by (Val Pack<:)

  E,Y,X<:Y ⊢ $\exists(Z<:X)A'(Z) <: \exists(Z<:Y)A(Z)$ as for (Sub Self)

  E ⊢ fold(A, pack Y<:A=C, $b\{Y\}:A(Y)$) ↔

  fold(A', pack Y<:A'=C, $b'\{Y\}:A'(Y)$) : A by (Eq Fold<: Lemma)

- (Eq Sub ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

  E, $x_i:A'(C)$ ⊢ $b_i\{C\} : B_i\{C\}$ $\forall i \in 1..n+m$

  by assumptions and a standard weakening lemma, using A'(C)<:A(C)

  E ⊢ $[l_i=\varsigma(x_i:A'(C))b_i\{C\}^{\ i\in 1..n+m}] : A'(C)$ by (Val Object)

$E \vdash [l_i=\varsigma(x_i:A(C))b_i\{C\}^{\ i\in 1..n}] \leftrightarrow$
$[l_i=\varsigma(x_i:A'(C))b_i\{C\}^{\ i\in 1..n+m}] : A(C)$ by (Eq Sub Object)

$E \vdash \varsigma(Y<:A=C)[l_i=\varsigma(x_i:A(Y))b_i\{Y\}^{\ i\in 1..n}] \leftrightarrow$
$\varsigma(Y<:A'=C)[l_i=\varsigma(x_i:A'(Y))b_i\{Y\}^{\ i\in 1..n+m}] : A$ by (Eq Self Lemma)

- (Eval ςSelect) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c\equiv\varsigma(Z<:A=C)a\{Z\}$)
  $E, Y<:A, y:A(Y) \vdash y.l_j : B_j\{A^+\}$ as for (Val ςSelect)
  $E \vdash$ (use a as $Y<:A, y:A(Y)$ in $y.l_j:B_j\{A^+\}$) $\leftrightarrow$ $a\{C\}.l_j : B_j\{A^+\}$ by (Eval Unself)

- (Eval ςOverride) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c\equiv\varsigma(Z<:A=C)a\{Z\}$)
  $E, Y<:A, y:A(Y) \vdash \varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,y,x\} : A$ as for (Val ςOverride)
  $E \vdash$ (use c as $Y<:A, y:A(Y)$ in $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,y,x\}:A$)
  $\leftrightarrow \varsigma(Z<:A=C)\ a\{C\}.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,a\{C\},x\}$ by (Eval Unself)
  $\leftrightarrow \varsigma(Z<:A=C)\ a\{Z\}.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,a\{Z\},x\}$: A by (Eq Self)

7.2 Four Examples, with Typing

We are now ready to reexamine the examples of section 2.4. We find that these examples can be typed rather easily when seen in terms of ς-objects, even when a method needs to return or to modify self. When constructing a fixed ς-object, its methods are not required to operate on an arbitrary self: they just need to match the given representation type of the object being constructed. That is, to construct a ς-object of type $A\equiv\varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$ we need only a set of methods $b_j:B_j\{A^+\}$ (not $b_j:B_j\{X^+\}$ for an arbitrary $X<:A$). Moreover, each of these methods can assume the existence a self parameter $x_j:[l_i:B_i\{A^+\}^{\ i\in 1..n}]$. (See the rules (Val ςObject) and (Val Object).)

7.2.1 Movable Points

This is the typed version of the example in section 2.4.1. We begin by defining the types of one-dimensional and two-dimensional movable points via the Self quantifier:

$P_1 \triangleq \varsigma(Self^+)[x:Int, mv_x:Int\to Self]$
$P_2 \triangleq \varsigma(Self^+)[x,y:Int, mv_x,mv_y:Int\to Self]$

We have the long sought property $P_2 <: P_1$, by (Sub ςObject).

Next we define the one-dimensional origin point, where Self names the type of the object, P_1:

$origin_1 \triangleq \varsigma(Self=P_1)\ [x=0, mv_x=\varsigma(s:P_1(Self))\lambda(dx:Int)\varsigma\langle Self, s.x:=s.x+dx\rangle]$

The typing $origin_1:P_1$ can be derived as follows, with $P_1(P_1)\equiv[x:Int, mv_x:Int\to P_1]$ as the chosen representation type for P_1. Recall that $\varsigma\langle P_1,c\rangle$ abbreviates $\varsigma(X<:P_1=P_1)c$ for an unused X:

$s:P_1(P_1), dx:Int \vdash s.x:=s.x+dx : P_1(P_1)$ by (Val Select), (Val Override)
$s:P_1(P_1), dx:Int \vdash \varsigma\langle P_1, s.x:=s.x+dx\rangle: P_1$ by (Val ςObject)
$s:P_1(P_1) \vdash \lambda(dx:Int)\varsigma\langle P_1, s.x:=s.x+dx\rangle: Int\to P_1$ by (Val Fun)
$\vdash [x=0, mv_x=\varsigma(s:P_1(P_1))\lambda(dx:Int)\varsigma\langle P_1, s.x:=s.x+dx\rangle] : P_1(P_1)$ by (Val Object)
$\vdash \varsigma(Self=P_1)\ [x=0, mv_x=\varsigma(s:P_1(Self))\lambda(dx:Int)\varsigma\langle Self, s.x:=s.x+dx\rangle] : P_1$ by (Val ςObject)

The rule (Val ςSelect) allows us to invoke methods whose type involves Self:

$\text{origin}_1.\text{mv_x} : \text{Int} \to P_1$

Moreover, the equational theory allows us to derive expected equivalencies, such as:

$$\text{origin}_1.\text{mv_x}(1) \leftrightarrow \varsigma(\text{Self}=P_1)\,[x=1,\ \text{mv_x}=\varsigma(s:P_1(\text{Self}))\lambda(dx:\text{Int})\varsigma\langle \text{Self},\ s.x:=s.x+dx\rangle] : P_1$$

that is, the unit point equals the result of moving the origin point.

As an aside, we comment on the so-called *binary methods*. A typical example is an equality method in a point object:

$$\begin{aligned} P_{1eq} &\triangleq \varsigma(\text{Self}^+)[x:\text{Int},\ \text{mv_x}:\text{Int}\to\text{Self},\ \text{eq}:\text{Self}\to\text{Bool}] \\ \text{origin}_{1eq} &\triangleq \varsigma(\text{Self}=P_{1eq})\,[x=0,\ \text{mv_x}=...,\ \text{eq} = \varsigma(s:P_{1eq}(\text{Self}))\ \lambda(p:\text{Self})\ s.x =_{\text{Int}} p.x] \end{aligned}$$

Binary methods violate the covariance requirement of ς-object types. We shall temporarily ignore this requirement, which is just a convention. We use the rules for the ς quantifier, which do not depend on covariance, instead of the derived rules for ς-objects.

We discover that binary methods cannot be invoked effectively because of typing restrictions. Expanding the encoding of method invocation, from section 7.1, we may try to pull out the eq method from a ς-object p of type P_{1eq}:

$$\text{use p as Self}<:P_{1eq},\ z:[x:\text{Int},\ \text{mv_x}:\text{Int}\to\text{Self},\ \text{eq}:\text{Self}\to\text{Bool}]\ \text{in z.eq} \qquad \text{(not typable)}$$

The rule (Val Use) requires that z.eq be given a type independent of X. This is not possible because X is in contravariant position in X→Bool.

Although we cannot pull out the eq method from p, we can still apply it within the scope of the use construct, if we can find an adequate argument. One possibility is:

$$\text{use p as Self}<:P_{1eq},\ z:[x:\text{Int},\ \text{mv_x}:\text{Int}\to\text{Self},\ \text{eq}:\text{Self}\to\text{Bool}]\ \text{in z.eq(z.mv_x(1))} \ :\ \text{Bool}$$

which compares p with its translation by 1, returning false.

However, most advantageous uses of eq would involve comparisons with independently obtained points, and these comparisons are not possible. This situation arises, essentially, because independent instances of the same existential type do not intermix.

Binary methods have proven very hard to handle in general, and we have little to contribute.

7.2.2 Backup Methods

This is the typed version of the example in section 2.4.2. The basic type in question is:

$$Bk \triangleq \varsigma(\text{Self}^+)[\text{retrieve}:\text{Self},\ \text{backup}:\text{Self}]$$

To show a less trivial example of backup, we define backed-up movable points:

$$BkP_1 \triangleq \varsigma(\text{Self}^+)[\text{retrieve}:\text{Self},\ \text{backup}:\text{Self},\ x:\text{Int},\ \text{mv_x}:\text{Int}\to\text{Self}]$$

Then we have both $BkP_1 <: P_1$ and $BkP_1 <: Bk$. Using the same techniques as in the previous example we obtain the following typing:

$$\begin{aligned} &\varsigma(\text{Self}=BkP_1) \\ &\quad [\text{retrieve} = \varsigma(s:BkP_1(\text{Self}))\varsigma\langle \text{Self},s\rangle, \\ &\quad\ \text{backup}=\varsigma(s:BkP_1(\text{Self}))\varsigma\langle \text{Self},\ s.\text{retrieve}\Leftarrow\varsigma(s':BkP_1(\text{Self}))\varsigma\langle \text{Self},s\rangle\rangle, \\ &\quad\ x = ...,\ \text{mv_x} = ...] \ :\ BkP_1 \end{aligned}$$

7.2.3 Object-Oriented Natural Numbers

This is the typed version of the example in section 2.4.3. The type of natural numbers is quite interesting: it contains a universal quantification of the form $\forall(Z)Z\to(X\to Z)\to Z$ with a free X. Note that X is in covariant position, although not in a strictly-positive position:

$$N_{Ob} \triangleq \varsigma(Self^+)[succ:Self,\ case:\forall(Z)Z\to(Self\to Z)\to Z]$$

The zero numeral can then be typed as follows:

$$\begin{array}{l} zero_{Ob} : N_{Ob} \triangleq \\ \quad \varsigma(Self=N_{Ob}) \\ \qquad [\ case = \lambda(Z)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ z, \\ \qquad succ = \varsigma(n:N_{Ob}(Self))\ \varsigma\langle Self,\ n.case := \lambda(Z)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ f(\varsigma\langle Self,n\rangle)\rangle\ ] \end{array}$$

The operations are:

$$\begin{array}{rcll} succ & : & N_{Ob}\to N_{Ob} & \triangleq\ \lambda(n:N_{Ob})\ n.succ \\ pred & : & N_{Ob}\to N_{Ob} & \triangleq\ \lambda(n:N_{Ob})\ n.case(N_{Ob})(zero_{Ob})(\lambda(p:N_{Ob})p) \\ iszero & : & N_{Ob}\to Bool & \triangleq\ \lambda(n:N_{Ob})\ n.case(Bool)(true)(\lambda(p:N_{Ob})false) \end{array}$$

Looking back at section 5.3, we can now explain the correspondence between our second-order typing for numerals and the typing based on sums. The type for the case method, Unit+Self, can be encoded as $\forall(Z)Z\to(Self\to Z)\to Z$, since in general A+B can be encoded as $\forall(Z)(A\to Z)\to(B\to Z)\to Z$. Using this encoding, we can view the definition of section 5.3 as a variant of the one given here.

7.2.4 A Calculator

This is the typed version of the example in section 2.4.4. Once the code is properly annotated, the calculator has type:

$$C = \varsigma(Self^+)[arg,acc:\ Real,\ enter:\ Real\to Self,\ add,sub:\ Self,\ equals:\ Real]$$

By subsumption, the calculator also has type:

$$Calc = \varsigma(Self^+)[enter:\ Real\to Self,\ add,sub:\ Self,\ equals:\ Real]$$

This shorter Calc type is the one shown to users of the calculator. As a common pattern in the code below, methods for a ς-object type A use the construction $\varsigma\langle Self,c\rangle$ to produce a value of type Self from a value c of type A(Self).

$$\begin{array}{l} calculator: C \triangleq \\ \quad \varsigma(Self=C) \\ \qquad [arg = 0.0, \\ \qquad\ acc = 0.0, \\ \qquad\ enter = \varsigma(s:C(Self))\ \lambda(n:Real)\ \varsigma\langle Self,\ s.arg := n\rangle, \\ \qquad\ add = \varsigma(s:C(Self))\ \varsigma\langle Self,\ (s.acc := s.equals).equals \Leftarrow \varsigma(s':C(Self))\ s'.acc+s'.arg\rangle, \\ \qquad\ sub = \varsigma(s:C(Self))\ \varsigma\langle Self,\ (s.acc := s.equals).equals \Leftarrow \varsigma(s':C(Self))\ s'.acc-s'.arg\rangle, \\ \qquad\ equals = \varsigma(s:C(Self))\ s.arg \\ \qquad\ ] \end{array}$$

A scientific calculator could also be defined, with additional state and operations. Its inner design might be quite different from that of our basic calculator, but the scientific calculator's type could still be a subtype of Calc.

7.3 Classes and Inheritance, with Self

We now extend the techniques developed in section 4.3 to model classes and inheritance. We take into account Self and, therefore, the need for pre-methods and classes to be parametric.

If $A \equiv \varsigma(Self^+)[l_i:B_i\{Self\}^{\,i\in 1..n}]$ is an object type, then:

$$Class(A) \triangleq [l_i:\forall(Self<:A)A(Self)\rightarrow B_i\{Self\}^{\,i\in 1..n}]$$

can be seen as a class type. This is the type of a collection of pre-methods that can be used in objects of type A and reused in other classes. Each pre-method is required to work for an arbitrary subtype Self of A; in particular, it is required to produce results that are parametric in Self.

As before, we can check easily that A<:A' does not imply Class(A)<:Class(A'). Hence, we define:

$$Class(A) \le Class(A') \quad \text{iff} \quad A <: A' \qquad (\le \text{ means "may inherit from"})$$

If $Class(A) \le Class(A')$, then we have $A \equiv \varsigma(Self^+)[l_i:B_i\{Self\}^{\,i\in 1..n+m}]$ and $A' \equiv \varsigma(Self^+)[l_i:B_i\{Self\}^{\,i\in 1..n}]$. So, $\forall(Self<:A')A'(Self)\rightarrow B_i\{Self^+\} <: \forall(Self<:A)A(Self)\rightarrow B_i\{Self^+\}$ for $i \in 1..n$. Hence, pre-methods of members of Class(A') may be reused for members of Class(A). Note that this reuse property would not hold if we took $A'\rightarrow B_i\{A'^+\}$ and $A\rightarrow B_i\{A^+\}$ as the types of the pre-methods, that is, if the pre-methods were not parametric in Self.

Finally, given a class of type Class(A), we can write a function $new_A : Class(A) \rightarrow A$ that generates objects from that class.

$$new_A \triangleq \lambda(c:Class(A))\ \varsigma(Self=A)\ [l_i=\varsigma(s:A(Self))\ c.l_i(Self)(s)^{\,i\in 1..n}]$$

As in section 4.3, we can embed new into classes, defining:

$$Class^{new}(A) \quad \triangleq \quad [new:A, l_i:\forall(Self<:A)A(Self)\rightarrow B_i\{Self\}^{\,i\in 1..n}]$$

There remains a serious problem in defining classes: writing sufficiently parametric pre-methods of types $\forall(Self<:A)A(Self)\rightarrow B_i\{Self^+\}$. Solutions are discussed in section 8.2.1 and 8.3.

8. Overriding and Self

In this section we discuss attempts to override methods that return self.

If we want to override a method of a ς-object of type A, the new method must work for any possible subtype of A. This is because the ς-object might have been constructed as an element of an unknown proper subtype of A. If the new method returns self, it is critical that the type of its result be the unknown subtype of A, because one of the other methods may be invoked on the result. We say that the overriding method must be "parametric in self "; this turns out to be a difficult criterion to meet.

It should not be too surprising that it is hard to override methods that return self. After all, the technique for obtaining ς-object subtypings is based on that of section 6.4 for obtaining covariant object components. As discussed there, covariant components cannot be usefully overridden.

8.1 Overriding from the Outside

At the beginning of section 7.2 we remark how easy it is to create a ς-object, because its initial methods need to work only for the actual type of the object being constructed. In particular, methods that override self present no difficulties.

However, if we want to override a method of an existing ς-object a:A, the new method must work for any possible B<:A, because a might have been built as an element of B. We do not know either the "true type" of a, or the "true type" of the self parameters of its methods. When overriding a method of a, the overriding method can assume only that the object has been constructed from an unknown Self<:A. The same difficulty would likely surface at object-creation time, if we were creating objects incrementally, adding methods to an empty object, instead of creating full objects at once.

This is where we need the complex rule for overriding ς-objects, (Val ςOverride), which was not used in section 7.2 even when typing override operations. Consider, for example, the type:

$$A \triangleq \varsigma(Self^+)[n,f:Int,\ m:Self] \qquad \text{with} \qquad A(X) \equiv [n,f:Int,\ m:X]$$

An overriding method for a≡ς(Y<:A=C)b{Y} can use in its body the variables Self<:A, x':A(Self), and x:A(Self), where x' is in fact b{C}, according to (Eval ςOverride), and x is the self of the new method. We can therefore override the f method of A with any of the following ways, with methods that produce integers.

a.f ⇐ ς(Self<:A,x':A(Self),x:A(Self)) 3 — setting a.f to produce 3 (constantly)

a.f ⇐ ς(Self<:A,x':A(Self),x:A(Self)) x'.n + 1 — setting a.f to produce b.n + 1 (constantly)

a.f ⇐ ς(Self<:A,x':A(Self),x:A(Self)) x.n + 1

setting a.f to produce 1 + the value of n when f is invoked

a.f ⇐ ς(Self<:A,x':A(Self),x:A(Self)) (x.n:=x'.n).n

setting a.f to update n with b.n (constantly) when f is invoked

Let us now attempt to override the m method. The typing rule (Val ςOverride) requires that, from the variables Self<:A, x':A(Self), x:A(Self) at its disposal, the overriding method must produce a value of type Self. Here are some possibilities:

a.m ⇐ ς(Self<:A,x':A(Self),x:A(Self)) x'.m

setting a.m to produce the current b.m (constantly)

a.m ⇐ ς(Self<:A,x':A(Self),x:A(Self)) x.m

setting a.m to diverge

However, we cannot override a.m with anything useful. Note, first, that we cannot synthesize a value of type Self from scratch. Second, we cannot return x' or x, nor ς⟨A,x'⟩ or ς⟨A,x⟩, because none of these can be given type Self. Third, ς⟨Self,x⟩ : Self is not derivable, for an unknown Self<:A; see (Val ςObject). Finally, any update to x' or x will preserve their original type, so the updated x' or x cannot be returned either.

Moreover, it would be unsound to ignore these typing problems and return, say, ς⟨Self,x⟩ or ς⟨Self,x'⟩. The reason can be seen in the following example, which builds a ς-object r_2 from a proper subtype R_1 of its own type R_2:

$R_2 \triangleq \varsigma(Self^+)[p:Self, q:Int]$

$R_1 <: R_2 \triangleq \varsigma(Self^+)[p:Self, q:Int, t:Int]$

$r_1 : R_1 \triangleq \varsigma(Y<:R_1=R_1)\ [p=\varsigma(s:R_1(Y))\varsigma\langle Y,s\rangle,\ q=0,\ t=0]$

$r_2 : R_2 \triangleq \varsigma(Y<:R_2=R_1)\ [p=r_1,\ q=\varsigma(s:R_2(Y))s.p.t]$

$r_2.p \Leftarrow \varsigma(Self<:R_2, x':R_2(Self), x:R_2(Self))\ \varsigma\langle Self,x\rangle$

$= \varsigma(Y<:R_2=R_1)\ [p=\varsigma(x:R_2(Y))\varsigma\langle Y,x\rangle,\ q=\varsigma(s:R_2(Y))\ s.p.t]$ (unsound)

$r_2.p \Leftarrow \varsigma(Self<:A, x':A(Self), x:A(Self))\ \varsigma\langle Self,x'\rangle$

$= \varsigma(Y<:R_2=R_1)\ [p=\varsigma(x:A(Y))r_2,\ q=\varsigma(s:R_2(Y))\ s.p.t]$ (unsound)

Unsound behavior can be observed by invoking q after either of the two updates above, because the field t is missing from s.p.

The reason for this unsound behavior can be traced back to the rule for constructing ς-objects. Because of the flexibility we have in constructing ς-objects out of proper subtypes of themselves, the x and x' parameters at our disposal when overriding may be shorter than the subtype used originally when constructing the object. For example, r_2 is constructed with a component p that is longer than the body of r_2, to which x or x' would be bound. Therefore, returning x or x' would not be safe, because they would be too short.

In conclusion, we discover that the (Val ςOverride) rule, although very powerful for overriding simple methods and fields, is not sufficient to allow us to override methods that return a value of type Self. We are in the strange but apparently inevitable position of being able to construct a point with a mv_x method, but unable then to override mv_x with an identical method. One solution to this problem is discussed in the next section.

8.2 Recoup

In this section we introduce a special method called *recoup* with an associated run-time invariant. Recoup is a method that returns self immediately. The invariant asserts that the result of recoup is its host object. These simple assumptions have a number of surprising consequences.

8.2.1 The Recoup Method

Let us consider the type:

$R \triangleq \varsigma(Self^+)[r:Self]$

We can build an element of R as follows:

$E, s:[r:R] \vdash s : [r:R]$	(Val x)
$E, s:[r:R] \vdash \varsigma(Z<:R=R)s : R$	(Val Self)
$E, s:[r:R] \vdash \varsigma\langle R,s\rangle : R$	(standard abbreviation)
$E \vdash [r = \varsigma(s:[r:R])\varsigma\langle R,s\rangle] : [r: R]$	(Val Object)
$E \vdash \varsigma(Self<:R=R)\ [r = \varsigma(s:[r:Self])\varsigma\langle Self,s\rangle] : R$	(Val Self)
$E \vdash \varsigma(Self<:R=R)\ [r = \varsigma(s:R(Self))\varsigma\langle Self,s\rangle] : R$	(standard abbreviation)

We say that a method of the form $\varsigma(s:A(Self))\varsigma\langle Self,s\rangle$, in the context of a ς-object of the form $\varsigma(Self<:A=A)[...]$, is a recoup method.

We may now revisit the backup example:

$$Bk \triangleq \varsigma(Self^+)[retrieve:Self, backup:Self]$$

As we have just discussed, we cannot usefully override the retrieve or backup methods, or any method with result type Self. This is because we are not able to produce a useful element of type Self from an element y : [retrieve:Self, backup:Self] as would be required by the (Val ςOverride) rule, other than y.retrieve and y.backup. We now introduce a technique that solves this problem.

Let us add a new component, called recoup, to the type Bk, and consider objects where recoup is set to $\varsigma(s:Bk(Bk))\varsigma\langle Bk,s\rangle$. We redefine:

$$Bk \triangleq \varsigma(Self^+)[recoup:Self, retrieve:Self, backup:Self]$$

$$a : Bk \triangleq \varsigma(Self=Bk)\ b$$
$$\text{where}\quad b \equiv [recoup=\varsigma(s:Bk(Self))\varsigma\langle Self,s\rangle,$$
$$retrieve=\varsigma(s:Bk(Self))\varsigma\langle Self,s\rangle,$$
$$backup=\varsigma(s:Bk(Self))\varsigma\langle Self,\ s.retrieve:=\varsigma\langle Self,s\rangle\rangle\ ]$$

Then, the following method update typechecks, since s'.recoup has type Self:

$$a.retrieve \Leftarrow \varsigma(Self<:Bk,\ s':Bk(Self),\ s:Bk(Self))\ s'.recoup$$

Moreover, the behavior obtained is a useful one, corresponding to performing a backup "by hand" instead of calling the backup method to do it. Omitting some type information, we calculate:

$$(\varsigma(Self=Bk)\ b).retrieve \Leftarrow \varsigma(Self<:Bk,s':Bk(Self),s:Bk(Self))\ s'.recoup$$
$$\leftrightarrow \varsigma(Self=Bk)\ b.retrieve \Leftarrow \varsigma(s:Bk(Self))\ b.recoup$$
$$\leftrightarrow \varsigma(Self=Bk)\ b.retrieve \Leftarrow \varsigma(s:Bk(Self))\ a$$
$$\leftrightarrow \varsigma(Self=Bk)\ [recoup=\varsigma(s:Bk(Self))\varsigma\langle Self,s\rangle,\quad retrieve=a,\quad backup=...]$$

Similarly, the following override clears the backup log:

$$a.retrieve \Leftarrow \varsigma(Self<:Bk,s':Bk(Self),s:Bk(Self))\ s.recoup$$

Intuitively, recoup allows us to recover a "parametric self " b.recoup, which equals a but has type Self<:Bk and not just type Bk. This technique is particularly useful after an override on a value of type Self<:Bk (as in the backup method), because the result of the override only has type Bk.

In general, if A has the form $\varsigma(Self^+)[recoup:Self, ...]$ then we can write useful polymorphic functions of type:

$$\forall(Self<:A)\ A(Self)\rightarrow Self$$

that are not available without recoup, for example:

$$f : \forall(Self<:Bk)\ Bk(Self)\rightarrow Self \triangleq$$
$$\lambda(Self<:Bk)\ \lambda(s:Bk(Self))\ (s.retrieve:=s.recoup).recoup$$

Such functions are parametric enough to be used in overrides from the outside, as in:

$$a.retrieve \Leftarrow \varsigma(Self<:Bk,\ s':Bk(Self),\ s:Bk(Self))\ f(Self)(s)$$

and can also be used for defining classes (see section 7.3).

8.2.2 The Recoup Invariant

The technique just described gives the correct result only as long as recoup is bound to ς(s:A⟨Self⟩)ς⟨Self,s⟩. Otherwise, the operational behavior is not the expected one. We must assume that recoup is bound to ς(s:A⟨Self⟩)ς⟨Self,s⟩ as an invariant. We have no elegant way of enforcing this invariant, since recoup is a component like any other. Furthermore, our ability to reason about programs is dependent on the recoup invariant:

a.backup ⇐ ς(Self,s',s) (s.retrieve:=s.recoup).recoup (identical override of a.backup)
↔ ς(Self=Bk) b.backup ⇐ ς(s:Bk⟨Self⟩)(s.retrieve:=s.recoup).recoup
↔ ς(Self=Bk) b.backup ⇐ ς(s:Bk⟨Self⟩)(s.retrieve:=ς⟨Self,s⟩).recoup
(only assuming s.recoup↔ς⟨Bk,s⟩)
↔ ς(Self=Bk) b.backup ⇐ ς(s:Bk⟨Self⟩)ς⟨Self, s.retrieve:=ς⟨Self,s⟩⟩
(only assuming (s.retrieve:=ς⟨Self,s⟩).recoup↔ς⟨Self, s.retrieve:=ς⟨Self,s⟩⟩)
↔ a

The correctness of typing, on the other hand, does not depend on the recoup invariant.

An invariant of this kind is, we believe, perfectly acceptable for a programming language: recoup would be a distinguished component that is appropriately initialized and that cannot be overridden. This option is investigated in the next section.

Even without language support, we may be disciplined enough to preserve the recoup invariant, and thus we may solve the problem of overriding methods with result type Self. If we initialize recoup correctly and do not override it, then the operational semantics will work out as expected.

8.2.3 Recoup as the Only True Method

Despite its simple code, recoup has remarkable power: it is the only proper method one ever needs. An object with recoup need not contain any other proper methods, only fields. For example, suppose we have:

A ≜ [m:B]
a : A ≜ [m=ς(s:A)b]

We rewrite A and a, introducing a Self quantifier and a recoup method r, and changing the method m into a field m' containing a function. This modification requires using the general Self quantifier without covariance restrictions (from section 6.5). We take:

A' ≜ ς(Self)[r:Self, m':Self→B]
a' : A' ≜ ς(Self=A') [r=ς(s:A'⟨Self⟩)ς⟨Self,s⟩, m'=λ(s:Self)b']

Thus the proper method m=ς(s:A)b in a is replaced with the field m'=λ(s:A'⟨Self⟩)b' in a'. Next we consider how to simulate operations on elements of A with operations on elements of A', and specifically how to encode invocation and override. Using our encoding of invocation and override, we can find a b' that imitates b.

As already noted a'.m' cannot be given a type. (Self must not escape the scope of ς, and the contravariant occurrence of Self, unlike the covariant ones, cannot be replaced by its bound.) Still, we can extract a'.m' inside the scope of ς and immediately apply it to a'.r, thus eliminating the single contravariant occurrence of Self before exiting the scope of the quantifier:

invoke(a', m') ≜ use a' as Y<:A', y:A'⟨Y⟩ in y.m'(y.r) : B

If a'.r is ς(s:A'⟨A'⟩)ς⟨A',s⟩ as expected, then invoke(a', m') has the same behavior as a.m.

Furthermore, m' can be overridden with any function f: $\forall(Y<:A')Y\rightarrow B$:

override(a', m', f) $\triangleq$ use a' as Y<:A', y:A'⟨Y⟩ in ς(X=Y) y.m':=f(X) : A'

More general override constructs are available as well, because what used to be methods can now be extracted as functions within the scope of an override. For example, when we have a movable point a' of type ς(Self)[r:Self, mv_x': Self→Int→Self, ...] we can override the move method (a function field) with one that executes the old method with a different parameter:

use a' as Y<:A', y:[r:Y, mv_x': Y→Int→Y, ...]
in ς(Self=Y) y.mv_x':=λ(s:Self)λ(n:Int)y.mv_x'(s)(n+1)
: ς(Self)[r:Self, mv_x': Self→Int→Self, ...]

Thus we can reuse the code of old methods, and not just their results, in defining new methods. This ability is reminiscent of that provided by *super* and *method wrappers* facilities [Weinreb, Moon 1981], and by the elder construct of section 4.5.3.

We can thus replace methods with function fields, opening some interesting possibilities. While we have done this for an object type A without Self, our technique might work even for ς-object types. Still, we cannot apply the technique to the recoup method, because we need recoup as a method to define invoke(a',m').

8.3 Objects with Structural Invariants

So far, we have worked with type and equational theories that are valid in standard denotational models, with a standard interpretation of subtyping (see section 9). However, if we imagine a programming language including our objects, we see that additional important properties may be satisfied by the operational semantics of the language that are not satisfied in standard models. These properties are, informally:

1) Structural subtyping invariant:
   every subtype of an object type is a longer object type
2) Object shape invariant:
   all elements of an object type have a certain shape

An example of a shape invariant, from section 8.2, is that all objects have a recoup method of a given form. In this section we study a different shape invariant, along with its typing and equational theory. Both structural subtyping and shape invariants must be assumed to achieve the desired effect.

Structural invariants have important consequences for typing, as we already discussed at the end of section 6.1. In particular, the structural invariants discussed in this section support stronger type rules for objects with Self. We call "structural" the rules that exploit the structural invariants. Most of the examples discussed so far can already by typed with the standard rules, but the stronger structural rules provide more natural typings. Other examples that could be typed only using recoup can now be typed with the structural rules, again in a more natural way.

In this section we provide plausibility arguments as to why these rules may be operationally sound. We have not constructed a model satisfying our structural rules.

Our goal is to define a new language with "structural object types" of the form $Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and "structural objects" of the form $obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n}]$. The language supports the stronger structural rules. To justify these rules, we provide an interpretation of structural objects into our ς-objects that establishes a particular shape invariant, namely that the interpretation of every structural object has a certain shape. Moreover, we provide an interpretation of the operations on

structural objects that preserves the invariant under reduction. In addition, if the result of a reduction is an object, then it is expressible as a structural object; so the new language is closed under reduction.

We show, within our standard formal system, that the structural rules are derivable in the presence of side conditions about the shape of terms. Then, under the hypothesis that the side conditions are invariantly satisfied, we remove the side conditions; this step is intuitively justified by the closure of the language under reduction. The removal of the side conditions has the following effects: (A) where a structural object was previously required, we allow a variable of a certain object type to occur (assuming that all object values of that type have the required shape); (B) where an object type with a certain shape and bound was previously required, we allow a type variable with the same bound to occur (assuming that only object types may be substituted for type variables bounded by object types).

8.3.1 Rules with Structural Side Conditions

We define a new syntax for object types, to indicate those types satisfying the structural subtyping invariant. We also define new syntax for objects, to indicate those objects satisfying a particular shape invariant. We give new syntax for object operations as well; the new operations will take advantage of the shape invariant.

$Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$	structural object type
$obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n}]$	structural object
$a.l_j$	structural object invocation
$a.l_j \Leftarrow \varsigma(Y<:A,\ x:Y)b\{Y,x\}$	structural object override

We define:

For $C \equiv Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n}]$, and fresh x_i':

$C(D) \triangleq [l_i:B_i\{D\}\ ^{i\in 1..n}]$

$a(D) \triangleq [l_i=\varsigma(x_i':C(D))b_i\{D,\varsigma\langle D,x_i'\rangle\}\ ^{i\in 1..n}]$

Our intended interpretation of this new syntax is in terms of ς-objects:

$$Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}] \triangleq \varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in 1..n}]$$

$$obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n}] \triangleq \varsigma(X<:C=C)\ [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n}]$$
for $C \equiv Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$

$$a.l_j \triangleq a(C).l_j$$
for $C \equiv Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n}]$, $j \in 1..n$

$$a.l_j \Leftarrow \varsigma(Y<:A,\ x:Y)b\{Y,x\} \triangleq \varsigma(Y<:A=C)\ a(Y).l_j \Leftarrow \varsigma(x':C(Y))b\{Y,\varsigma\langle Y,x'\rangle\}$$
for $C \equiv Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n}]$, $j \in 1..n$

Note the special properties of the ς-objects that correspond to structural objects: the external and internal types of $\varsigma(X<:C=C)\ [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n}]$ are the same (C), and all occurrences of self (x_i') in the body of methods are in the context $\varsigma\langle C,x_i'\rangle$.

Thus, intuitively, the shape invariant is that every object has the form:

$$\varsigma(X<:C=C)\ [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n}]$$

and the structural subtyping invariant is that every subtype of $Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$ is a longer object type, of the form $Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n+m}]$.

Given these definitions, we can derive the rules:

Objects, under structural side conditions

(Type Object')

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i \in 1..n}{E \vdash Obj(X)[l_i:B_i\{X_i\}^{\,i\in 1..n}]}$$

(Sub Object')

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i \in 1..n+m}{E \vdash Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Val Object') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i\{A\} : B_i\{A^+\} \quad \forall i \in 1..n}{E \vdash obj(X=A)[l_i=\varsigma(x_i:X)b_i\{X\}^{\,i\in 1..n}] : A}$$

(Val Select') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j : B_j\{A^+\}}$$

(Val Override') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, Y<:A, x:Y \vdash b\{Y,x\} : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, x:Y)b\{Y,x\} : A}$$

(Eval Select') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{C,a\}: B_j\{A^+\}}$$

(Eval Override') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, Y<:A, x:Y \vdash b\{Y,x\} : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A,x:Y)b\{Y,x\} \leftrightarrow obj(X=C)[l_j=\varsigma(x:X)b\{X,x\}, l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in(1..n+m)-\{j\}}] : A}$$

The shape invariant is established by (Val Object'), assuming it is true of subcomponents. The invariant is assumed explicitly by (Val Select') for a particular term; and preserved if already true of subcomponents. The invariant is also assumed explicitly by (Val Override'), and preserved for the overridden object and its subcomponents. The invariant is preserved by reduction, as seen in (Eval Select') and (Eval Override').

We now verify the rules in detail.

Lemma 8.3.1-1

Let $C \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in 1..n+m}]$.
If $E \vdash a : A$ then $E \vdash C <: A$ and $E \vdash a(C) : C(C)$.
Moreover, if A has the form Obj(X)[...], then $E \vdash C(C) <: A(C)$.

Proof

Suppose that for $C \equiv Obj(X)[l_i:B_i\{X\}\ ^{i\in 1..n}]$ and $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x_i\}\ ^{i\in 1..n+m}]$ we have a derivation of $E \vdash a : A$ for some A. This derivation may end with a number of subsumption steps, but in front of those steps, by the shape of a, there must be a (Val Fold) step with conclusion $E \vdash a : C$. Since $E \vdash a : A$ is obtained from $E \vdash a : C$ by subsumption steps, we have $E \vdash C <: A$. The assumption of the (Val Fold) step is:

$$E \vdash (\text{pack } X<:C=C, [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n+m}]:C(X)) : \exists(X<:C)[l_i:B_i\{X\}\ ^{i\in 1..n}]$$

In turn, this judgment must have been derived through a number of subsumption steps, with assumption:

$$E \vdash (\text{pack } X<:C=C, [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n+m}]:C(X)) : \exists(X<:C)[l_i:B_i\{X\}\ ^{i\in 1..n+p}]$$
for some $p \in 0..m$

preceded by a (Val Pack<:) step with assumption:

$$E \vdash [l_i=\varsigma(x_i':C(C))b_i\{C,\varsigma\langle C,x_i'\rangle\}\ ^{i\in 1..n+m}] : [l_i:B_i\{C\}\ ^{i\in 1..n+p}]$$

From this, by subsumption, we obtain:

$$E \vdash [l_i=\varsigma(x_i':C(C))b_i\{C,\varsigma\langle C,x_i'\rangle\}\ ^{i\in 1..n+m}] : [l_i:B_i\{C\}\ ^{i\in 1..n}]$$

That is, $E \vdash a(C) : C(C)$.
Now assume that A has the form Obj(X)[...]; then $E \vdash C<:A$ must come from an application of (Sub Rec) and an application of (Sub Exists), with assumption:

$$E, Y<:Y', X<:Y \vdash C(X) <: A(X)$$

By a standard substitution lemma (setting Y,Y', and X all to C), we have also $E \vdash C(C) <: A(C)$.

□

Proposition 8.3.1-1

The rules with structural side conditions are derivable.

Proof

- The rules (Type Object') and (Sub Object') follow directly from the definitions.
- (Val Object') where $A \equiv \varsigma(X)[l_i:B_i\{X^+\}\ ^{i\in 1..n}]$

$E, x_i:A \vdash b_i\{A,x_i\} : B_i\{A^+\}$	by assumption
$E, x'_i:A(A), x_i:A \vdash b_i\{A,x_i\} : B_i\{A^+\}$	by weakening
$E, x'_i:A(A) \vdash \varsigma\langle A,x_i'\rangle : A$	by (Val Self)
$E, x'_i:A(A) \vdash b_i\{A,\varsigma\langle A,x_i'\rangle\} : B_i\{A^+\}$	by substitution
$E \vdash [l_i=\varsigma(x'_i:A(A))b_i\{A,\varsigma\langle A,x'_i\rangle\}\ ^{i\in 1..n}] : A(A)$	by (Val Object)
$E \vdash \varsigma(X<:A=A)\ [l_i=\varsigma(x'_i:A(X))b_i\{X,\varsigma\langle X,x'_i\rangle\}\ ^{i\in 1..n}] : A$	by (Val Self)

- (Val Select') where $A \equiv \varsigma(X)[l_i:B_i\{X^+\}\ ^{i\in 1..n}]$, $a \equiv \varsigma(X<:C=C)\ [l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}\ ^{i\in 1..n+m}]$

$E \vdash a(C) : A(C)$ and $E \vdash C <: A$	by Lemma 8.3.1-1
$E \vdash a(C).l_j : B_j\{C^+\}$	by (Val Select)
$E \vdash B_j\{C^+\} <: B_j\{A^+\}$	by covariance of $B_j\{X\}$ in X
$E \vdash a(C).l_j : B_j\{A^+\}$	by (Val Subsumption)

- (Val Override') where $A \equiv \varsigma(X)[l_i:B_i\{X^+\}^{\,i\in 1..n}]$, $a \equiv \varsigma(X<:C=C)\,[l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}^{\,i\in 1..n+m}]$

  $E \vdash a(C) : C(C)$; $E \vdash C <: A$; $E \vdash C(C) <: A(C)$ by Lemma 8.3.1-1

  $E, x:C \vdash b\{C,x\} : B_j\{C^+\}$ by substitution of C for Y in the assumption

  $E, x':C(C), x:C \vdash b\{C,x\} : B_j\{C^+\}$ by weakening

  $E, x':C(C), x:C \vdash \varsigma\langle C,x'\rangle : C$ by (Val Self)

  $E, x':C(C) \vdash b\{C,\varsigma\langle C,x'\rangle\} : B_j\{C^+\}$ by substitution of $\varsigma\langle C,x'\rangle$ for x

  $E \vdash a(C).l_j \Leftarrow \varsigma(x':C(C))b\{C,\varsigma\langle C,x'\rangle\} : C(C)$ by (Val Override)

  $E \vdash \varsigma(Y<:C=C)\,a(Y).l_j \Leftarrow \varsigma(x':C(Y))b\{Y,\varsigma\langle Y.x'\rangle\} : C$ by (Val Self)

  $E \vdash \varsigma(Y<:C=C)\,a(Y).l_j \Leftarrow \varsigma(x':C(Y))b\{Y,\varsigma\langle Y.x'\rangle\} : A$ by (Val Subsumption)

- (Eval Select') where $A \equiv \varsigma(X)[l_i:B_i\{X^+\}^{\,i\in 1..n}]$, $a \equiv \varsigma(X<:C=C)\,[l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}^{\,i\in 1..n+m}]$

  $E \vdash a(C) : A(C)$ and $E \vdash C <: A$ by Lemma 8.3.1-1

  $E \vdash a(C).l_j \leftrightarrow b_j\{C,\varsigma\langle C,a(C)\rangle\} : B_j\{C^+\}$ by (Eval Select)

  $E \vdash a(C).l_j \leftrightarrow b_j\{C,a\} : B_j\{C^+\}$ by (Eq Self)

  $E \vdash B_j\{C^+\} <: B_j\{A^+\}$ by covariance of $B_j\{X\}$ in X

  $E \vdash a(C).l_j \leftrightarrow b_j\{C,a\} : B_j\{A^+\}$ by (Eq Subsumption)

- (Eval Override') where $A \equiv \varsigma(X)[l_i:B_i\{X^+\}^{\,i\in 1..n}]$, $a \equiv \varsigma(X<:C=C)\,[l_i=\varsigma(x_i':C(X))b_i\{X,\varsigma\langle X,x_i'\rangle\}^{\,i\in 1..n+m}]$

  $E \vdash a(C) : C(C)$; $E \vdash C <: A$; $E \vdash C(C) <: A(C)$ by Lemma 8.3.1-1

  $E, x':C(C) \vdash b\{C,\varsigma\langle C,x'\rangle\} : B_j\{C^+\}$ as in the (Val Override') case

  $E \vdash a(C).l_j \Leftarrow \varsigma(x':C(C))b\{C,\varsigma\langle C,x'\rangle\}$ by (Eval Override)

  $\leftrightarrow [l_j=\varsigma(x':C(C))b\{C,\varsigma\langle C,x'\rangle\}, l_i=\varsigma(x_i':C(C))b_i\{C,\varsigma\langle C,x_i'\rangle\}^{\,i\in(1..n+m)-\{j\}}] : C(C)$

  $E \vdash \varsigma(Y<:C=C)\,a(Y).l_j \Leftarrow \varsigma(x':C(Y))b\{Y,\varsigma\langle Y,x'\rangle\}$ by (Eq Self)

  $\leftrightarrow \varsigma(Y<:C=C)\,[l_j=\varsigma(x':C(Y))b\{Y,\varsigma\langle Y,x'\rangle\}, l_i=\varsigma(x_i':C(Y))b_i\{Y,\varsigma\langle Y,x_i'\rangle\}^{\,i\in(1..n+m)-\{j\}}] : C$

  $E \vdash$ *(same)* : A by (Eq Subsumption)

□

So far we have dealt only with the shape invariant. To exploit the structural subtyping invariant, we consider the following version of the rules, which is trivially derivable from the previous one. The rules differ by a redundant assumption $E \vdash A<:A'$ and by the restriction of l_j to the labels of A'.

(Val Select") (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n'}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A<:A' \qquad j\in 1..n'}{E \vdash a.l_j : B_j\{A^+\}}$$

(Val Override") (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n'}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A<:A' \qquad E, Y<:A, x:Y \vdash b\{Y,x\} : B_j\{Y^+\} \qquad j\in 1..n'}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, x:Y)b\{Y,x\} : A}$$

(Eval Select") (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n'}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad j \in 1..n}{E \vdash a.l_j \;\leftrightarrow\; b_j\{C,a\}: B_j\{A^+\}}$$

(Eval Override") (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n'}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i\{X,x\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y<:A, x:Y \vdash b\{Y,x\} : B_j\{Y^+\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A,x:Y)b\{Y,x\} \;\leftrightarrow\; obj(X=C)[l_j=\varsigma(x:X)b\{X,x\},\ l_i=\varsigma(x_i:X)b_i\{X,x_i\}^{\,i\in(1..n+m)-\{j\}}] : A}$$

8.3.2 Rules with Structural Assumptions

The rules we have derived so far are valid, but are not fully general: the typing rule for $a.l_j$, for example, is applicable only to a's of a specific form and not to $x.l_j$ when x is a variable of an appropriate object type. As we discussed at the beginning of the section, we expect that an operational interpretation of the rules will satisfy the invariants, so that it will be sound to assume the invariants even for variables, and not just for terms of special form. Hence, removing the side conditions about the form of a subtype of an object type, and the side conditions about the shape of an object (except for evaluation rules), we obtain the rules:

Objects, under structural invariants

(Type Structural Object)

$$\frac{E,X \vdash B_i\{X^+\} \qquad \forall i \in 1..n}{E \vdash Obj(X)[l_i:B_i\{X_i\}^{\,i\in 1..n}]}$$

(Sub Structural Object)

$$\frac{E,X \vdash B_i\{X^+\} \qquad \forall i \in 1..n+m}{E \vdash Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Val Structural Object) (where $A \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i\{A\} : B_i\{A^+\} \qquad \forall i \in 1..n}{E \vdash obj(X=A)[l_i=\varsigma(x_i:X)b_i\{X\}^{\,i\in 1..n}] : A}$$

(Val Structural Select) (where $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad j \in 1..n}{E \vdash a.l_j : B_j\{A^+\}}$$

(Val Structural Override) (where $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y<:A, x:Y \vdash b\{Y,x\} : B_j\{Y^+\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, x:Y)b\{Y,x\} : A}$$

(Eval Structural Select) (where $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad j \in 1..n}{E \vdash a.l_j \;\leftrightarrow\; b_j\{C,a\} : B_j\{A^+\}}$$

(Eval Structural Override) (where $A' \equiv Obj(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $a \equiv obj(X=C)[l_i=\varsigma(x_i:X)b_i^{\,i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y<:A, x:Y \vdash b\{Y\} : B_j\{Y^+\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A,x:Y)b\{Y\} \;\leftrightarrow\; obj(X=C)[l_j=\varsigma(x:X)b\{X\},\ l_i=\varsigma(x_i:X)b_i^{\,i\in(1..n+m)-\{j\}}] : A}$$

These rules could be developed into a complete formal system, similar to **ςOb** of appendix D.

We illustrate the use of the structural rules by adapting the example of the object-oriented natural numbers. Compare this formulation with the one in section 7.2.3:

$$N_{Ob} \triangleq Obj(Self)[succ:Self,\ case:\forall(Z)Z\to(Self\to Z)\to Z]$$

$$zero_{Ob} : N_{Ob} \triangleq obj(Self=N_{Ob})[\ case = \lambda(Z)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ z,\ succ = \varsigma(n:Self)\ n.case := \lambda(Z)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ f(n)]$$

We can now override even methods that return a value of type Self, given a sufficiently parametric new method. For example, we can typecheck:

$$zero_{Ob}.succ \Leftarrow \varsigma(Self<:N_{Ob},\ n:Self)\ n.case := \lambda(Z)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ f(n) \ :\ N_{Ob}$$

using (Val Structural Override) to obtain "n.case:= ... : Self":

We finish with an example of *inheritable* methods, that is, of methods that are defined parametrically for all subtypes of a certain type, and then reused by instantiation to any of these subtypes. Either recoup or the structural rules are essential in giving a type to these parametric methods.

We define the types of movable points and construct the origin objects in the new style. Here inc_x is defined parametrically and used for building elements of both P_1 and P_2. We can view $origin_2$ as inheriting inc_x from the definition of inc_x originally intended for $origin_1$. (We do not claim that $origin_2$ inherits directly from $origin_1$.) The code of mv_x is identical in the two origin objects although Self refers to two different types; this can be seen as a form of "textual" inheritance.

$$P_1 \triangleq Obj(Self)[x:Int,\ mv_x:Self]$$

$$inc_x : \forall(Self<:P_1)\ Self \to Self \triangleq \lambda(Self<:P_1)\ \lambda(s:Self)\ s.x:=s.x+1$$

$$origin_1 \triangleq obj(Self=P_1)[x=0,\ mv_x = \varsigma(s:Self)\ inc_x(Self)(s)]$$

$$P_2 \triangleq Obj(Self)[x,y:Int,\ mv_x,mv_y:Int\to Self]$$

$$origin_2 \triangleq obj(Self=P_2)[x=0,\ mv_x = \varsigma(s:Self)\ inc_x(Self)(s),\ y=0,\ mv_y= \ldots\ ]$$

Moreover, we can now define other parametric functions of type $\forall(Self<:P_1)\ Self \to Self$, and use them to override mv_x in any object of a subtype of P_1.

$\text{inc2_x} : \forall(\text{Self}<:P_1)\ \text{Self} \to \text{Self} \triangleq$
$\lambda(\text{Self}<:P_1)\ \lambda(s:\text{Self})\ s.x:=s.x+2$
$\text{origin}_1.\text{mv_x} \Leftarrow \varsigma(\text{Self}<:P_1, s:\text{Self})\text{inc2_x}(\text{Self})(s)$
$\text{origin}_2.\text{mv_x} \Leftarrow \varsigma(\text{Self}<:P_2, s:\text{Self})\text{inc2_x}(\text{Self})(s)$

As discussed in section 6.1, however, the type $\forall(\text{Self}<:P_1)\text{Self}\to\text{Self}$ contains only approximants of the identity function in parametric models; clearly these models do not validate our structural invariants.

9. A Semantics for Objects

In this section we describe a semantics for our largest language, $\mathbf{FOb}_{<:\mu}$, with the expected equational theory (given by the fragments $\Delta_{=}$, $\Delta_{=\mathbf{x}}$, $\Delta_{=\to}$, $\Delta_{=\mathbf{Ob}}$, $\Delta_{=<:}$, $\Delta_{=\mathbf{Ob}<:}$, $\Delta_{=<:\mu}$, $\Delta_{=<:\forall}$, and $\Delta_{=<:\exists}$).This semantics is based on the metric approach of MacQueen, Plotkin, and Sethi [MacQueen, Plotkin, Sethi 1986]. More precisely, we follow Amadio [Amadio 1989] and Cardone [Cardone 1989] in the use of complete uniform pers and contractive functions on pers. We rely on their work for the semantics of $\boldsymbol{F}_{<:\mu}$. We extend their work with a treatment of abstract data types and of object types.

It might be possible to obtain a semantics for $\mathbf{FOb}_{<:\mu}$ with standard O-categorical methods [Smyth, Plotkin 1982] instead of metric methods. However, even the fragment $\boldsymbol{F}_{<:\mu}$ poses problems, as noted in [Abadi, Plotkin 1990]. We expect object types to give rise to further, interesting difficulties.

Many of the ideas in our semantics originate in the study of Baby Modula-3 [Abadi 1994]. See section 11.1 for a comparison.

9.1 The Untyped Universe

We interpret all our languages in an untyped universe. This section described the structure of this untyped universe. The assumptions on the universe are fairly standard and technical [MacQueen, Plotkin, Sethi 1986; Amadio 1989; Abadi, Plotkin 1990]. The impatient reader may wish to skip this section.

We need to have a complete partial order $(D,\sqsubseteq)$ such that:

- There is an increasing sequence p_n: $D\to D$ of continuous projections with finite range and with least upper bound the identity. Further, p_0 constantly equals $\perp$.
- There are strict, continuous embedding-retraction pairs (e,r) between D and each of $W_\perp$, $(D\to D)$, and $(L\to D)_\perp$.

$$\begin{array}{rcccl} W_\perp & \xrightarrow{e_w} & D & \xrightarrow{r_w} & W_\perp \\ (D\to D) & \xrightarrow{e_f} & D & \xrightarrow{r_f} & (D\to D) \\ (L\to D)_\perp & \xrightarrow{e_o} & D & \xrightarrow{r_o} & (L\to D)_\perp \end{array}$$

  where

  - W is a one-point set $\{*\}$; we view $*$ as the error value;
  - $(D\to D)$ is the complete partial order of continuous functions from D to D;
  - L is a countable set of labels $\{m_0, m_1, ...\}$, and $(L\to D)$ is the complete partial order of functions from L to D; this is roughly the set of records over these labels;
  - $X_\perp$ denotes the lifting of X.

We commonly view W, D→D, and L→D as subsets of D, and write $x = *$ for $x = e_w(*)$, $x \in (D\to D)$ for $x \in \text{range}(e_f)$, and $x \in (L\to D)$ for $x \in e_o(L\to D)$. When x, y ∈ D, m ∈ L, we write x(y) for $r_f(x)(y)$ and x(m) for $r_o(x)(m)$. Below we omit the various e's and r's.

- Let ; denote function composition and ↾ denote function restriction. For all i,

$$\begin{aligned} p_{i+1}(e_w(*)) &= e_w(*) \\ p_{i+1}(e_f(f)) &= e_f(p_i; f; p_i) && f \in D\to D \\ p_{i+1}(e_o(o)) &= e_o((o; p_i)\restriction\{m_0,...,m_i\}) && o \in L\to D \end{aligned}$$

Thus, omitting e's and r's, $p_{i+1}(*)$ is $*$; $p_{i+1}(f)(x)$ is $p_i(f(p_i(x)))$; and $p_{i+1}(o)(m_j)$ is $p_i(o(m_j))$ if $j\le i$, and ⊥ otherwise.

If x ∈ D is such that $p_n(x) = x$ for some n then x is finite, and the least n for which $p_n(x) = x$ is the rank of x. If x is finite and $\langle y_i\rangle$ is an increasing chain then $x \sqsubseteq \bigsqcup_i y_i$ implies that $x \sqsubseteq y_k$ for some k.

A suitable D can be constructed by applying the usual "limit of a sequence of iterates" method to solve the domain equation $D = W_\perp + (D\to D) + (L\to D)_\perp$, making sure that $p_i(e_o(L\to D))$ is always a finite set.

9.2 The Interpretation of Untyped Terms

Next we define the interpretation of untyped terms, giving a precise semantics for the untyped calculus, independently of any typing considerations. The interpretation of typed terms will be based on the interpretation of untyped terms.

We define the semantics function for terms:

$$[\![\ ]\!] : (V\to D)\to(E\to D)$$

where V is the set of variables and E the set of expressions. We call a mapping ρ in V→D an environment and write $[\![a]\!]_\rho$ for the semantics of a term a with an environment ρ. We write $\langle\langle m_1{=}v_1, ..., m_n{=}v_n\rangle\rangle$ for the function in (L→D) that maps m_1 to v_1, ..., m_n to v_n, and maps all other labels to $*$. If f is a function and l is in its domain, we write $f\langle l\leftarrow v\rangle$ for the function that maps l to v and is identical to f elsewhere; otherwise, $f\langle l\leftarrow v\rangle$ is ⊥.

With this notation, we give an interpretation for the untyped calculus:

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![\lambda(x)b]\!]_\rho &= \lambda(v)\,[\![b]\!]_{\rho\langle x\leftarrow v\rangle} \\ [\![b(a)]\!]_\rho &= \text{if } [\![b]\!]_\rho \in (D\to D) \\ &\quad \text{then } [\![b]\!]_\rho([\![a]\!]_\rho) \\ &\quad \text{else } * \\ [\![[m_i{=}\varsigma(x_i)c_i\ ^{i\in I}]]\!]_\rho &= \langle\langle m_i{=}[\![\lambda(x_i)c_i]\!]_\rho\ ^{i\in I}\rangle\rangle \\ [\![a.m]\!]_\rho &= \text{if } [\![a]\!]_\rho \in (L\to D) \text{ and } [\![a]\!]_\rho(m) \in (D\to D) \\ &\quad \text{then } [\![a]\!]_\rho(m)([\![a]\!]_\rho) \\ &\quad \text{else } * \\ [\![a.m \Leftarrow \varsigma(x)c]\!]_\rho &= \text{if } [\![a]\!]_\rho \in (L\to D) \\ &\quad \text{then } [\![a]\!]_\rho\langle m\leftarrow[\![\lambda(x)c]\!]_\rho\rangle \\ &\quad \text{else } * \end{aligned}$$

This definition is given in a metalanguage where $v \sqsubseteq V$ is a strict membership test, and where conditionals and conjunctions are strict and evaluated left to right, e.g., $[\![a.m \Leftarrow \varsigma(x)c]\!]_\rho = \bot$ if $[\![a]\!]_\rho = \bot$.

Note how the semantics turns ς's into λ's and objects into records. In a sense, then, it is possible to encode an object calculus in a record calculus. The catch is that the denotations of types will not be standard record types. Note also that overriding a method of an object does not produce an error if the object does not have the method in the first place. This definition of override simplifies our technical treatment.

The semantics is sound in that it validates the simple reduction rules for objects. Many of the more interesting equational rules for objects fail, however; we develop a typed semantics in order to validate those rules.

9.3 Types in the Untyped Universe

Having described an untyped model, we view the types as certain binary relations on this untyped model. Intuitively, if A is a type and R_A is the associated relation, then $(x,y) \in R_A$ means that x and y are equal elements of A. Subtyping is simply interpreted as relation containment.

After some preliminaries, section 9.3.2 introduces operations on binary relations and discusses contractiveness. Section 9.3.3 provides lemmas about the union operation, and sections 9.3.4 and 9.3.5 provides other basic semantic results about metric properties and objects.

9.3.1 Preliminaries

We will be dealing with binary relations over D. By convention, we will always mean binary relations that do not have $*$ in their domains. It is easy to show that all of our constructions preserve this property.

A per is a symmetric, transitive, binary relation on D. A binary relation X is uniform if u X v implies $(p_i(u))$ X $(p_i(v))$ for all i. A binary relation X is complete if $\bot X \bot$ and X is closed under limits of increasing sequences in the $\sqsubseteq$ order. A cuper is a complete uniform per. The set of all cupers is **CUPER**. Below, all types are interpreted as pers in **CUPER**. Throughout, the variables R, T, and S range over **CUPER**.

9.3.2 Semantic Definitions

First we describe some usual operations on pers.

The function-space operation is given by:

$$R \to T = \{(f,g) \in (D \to D) \times (D \to D) \mid \text{if } xRy \text{ then } f(x)Tg(y)\}$$

If $R, T \in$ **CUPER** then $R \to T \in$ **CUPER**.

We can calculate meets and joins for any family of cupers:

$$\begin{aligned} \sqcap_{i \in I} X_i &= \cap_{i \in I} X_i \\ \sqcup_{i \in I} X_i &= \mathcal{C}(\cup_{i \in I} X_i) \end{aligned}$$

where $\mathcal{C}(X)$ is the least cuper that contains X. (The cuper $\mathcal{C}(X)$ is always defined since the conditions that describe cupers are monotone closure conditions.) If $X_i \in$ **CUPER** for all $i \in I$ then $\sqcap_{i \in I} X_i \in$ **CUPER** and $\sqcup_{i \in I} X_i \in$ **CUPER**.

The distance between two cupers is 2^{-r}, where r is the minimum rank where the two cupers differ, and it is 0 if the two cupers are equal. The set of all cupers with this distance function is a complete metric space. Furthermore, by the Banach Fixpoint Theorem, if F is a contractive map between cupers then it has a unique fixpoint. This is the basis of a usual interpretation of recursive types. If F(S) is a contractive function in S on **CUPER**, then we write $\mu(S)F(S)$ for its unique fixpoint.

Amadio [Amadio 1989] has verified that the usual type constructors $\rightarrow$, $\sqcap$, and μ are contractive or nonexpansive as in the ideal model [MacQueen, Plotkin, Sethi 1986]. On the other hand, the properties of $\sqcup$ seem largely unexplored. As proved in Proposition 9.3.4-3, it turns out that $\sqcup$ is nonexpansive.

In order to give a semantics to object types, we define:

$$\langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle \quad = \quad \{(\bot,\bot)\} \cup \{(o,o') \in (L\rightarrow D) \times (L\rightarrow D) \mid \forall i\in I.\ (o(m_i), o'(m_i)) \in T_i\}$$

The type $\langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle$ should be viewed as a record type, with fields m_i and types T_i. If $T_i \in$ **CUPER** for all $i \in I$ then $\langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle \in$ **CUPER**. Uniformity holds because, if $(o,o') \in \langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle$ then $(p_o(o),p_o(o')) = (\bot,\bot) \in \langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle$, and either $(p_{j+1}(o)(m_i),p_{j+1}(o')(m_i)) = (p_j((o)(m_i)),p_j((o')(m_i)))$ or $(p_{j+1}(o)(m_i),p_{j+1}(o')(m_i)) = (\bot,\bot)$, so $(p_{j+1}(o),p_{j+1}(o')) \in \langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle$.

A type function of the form $F = \lambda(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$ is called a generator, and the set of generators is **Gen**. Generators are suggested by the imperfect generator semantics of section 1. At the level of types, generators are rather useful.

Given a generator, it can be written in the form $\lambda(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$ uniquely:

Proposition 9.3.2-1

Let $F(S) = \langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle = \langle\langle m_j : S\rightarrow R_j \ ^{j\in J}\rangle\rangle$ for any S. Then $I = J$ and $T_i = R_i$ for all i.

Proof

If $j \notin I$ but $j \in J$, then $\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$ contains a pair (r,r) such that $r(m_j) = *$, not allowed by $\langle\langle m_j : S\rightarrow R_j \ ^{j\in J}\rangle\rangle$; for example, let r be the record $\langle\langle m_j=*, m_{k,k\neq j}=\bot\rangle\rangle$. If $I = J$ but $(v,v') \notin T_i$ and $(v,v') \in R_i$, then $\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$ does not contain (r,r') but $\langle\langle m_j : S\rightarrow R_j \ ^{j\in J}\rangle\rangle$ does, where $r = \langle\langle m_i=\lambda(x)v, m_{k,k\neq i}=\bot\rangle\rangle$, $r' = \langle\langle m_i=\lambda(x)v', m_{k,k\neq i}=\bot\rangle\rangle$.

□

We say that $F = \lambda(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$ extends $G = \lambda(S)\langle\langle m_j : S\rightarrow T_j \ ^{j\in J}\rangle\rangle$ if $I \supseteq J$, and write $F \preccurlyeq G$. We set:

$$[[m_i:T_i \ ^{i\in I}]] \quad = \quad \sqcup \{\mu(S)F(S) \mid F \in \textbf{Gen}, F \preccurlyeq \lambda(S)\langle\langle m_i:S\rightarrow T_i \ ^{i\in I}\rangle\rangle\}$$

The following proposition shows that this definition is proper:

Proposition 9.3.2-2

If $F \in$ **Gen** then F(S) is contractive in S. Hence $\mu(S)F(S)$ exists and is unique.

Proof

This follows because $\rightarrow$ is contractive and $\sqcap$ is nonexpansive, and

$$\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle \quad = \quad \{(\bot,\bot)\} \cup \sqcap_{i\in I} \{(o,o') \in (L\rightarrow D) \times (L\rightarrow D) \mid (o(m_i), o'(m_i)) \in S\rightarrow T_i\}$$

□

The types of the form $[[m_i:T_i \ ^{i\in I}]]$ are our object types. In the imperfect self-application semantics of section 1, we attempt to model object types as recursive record types, but fail to obtain all the expected subtypings. Our semantic approach is based on that attempt. Here, however, we have unions available. We define an object type to be a union of recursive record types. We make the union large enough to obtain all the expected subtypings: each object type is designed to contain all longer object types.

A proof that the object-type construction is contractive appears in section 9.3.4.

9.3.3 Lemmas about Unions

In this section we obtain some necessary results about unions of relations. A union is defined with a closure operation. The definition does not give a very explicit description of the elements of the

union. In particular, it is not true that if $(x,y) \in S \sqcup T$ then either $(x,y) \in S$ or $(x,y) \in T$, and the definition of $S \sqcup T$ does not help much in pinning down what else (x,y) could be.

Here we analyze the closure operation. We obtain results that, for finite elements, reduce the closure operation to the familiar transitive closure operation (Propositions 9.3.3-8 and 9.3.3-9). In the end, we obtain lemmas that enable us to reason about all elements of a union by reasoning about the elements of the components of the union (Lemmas 9.3.3-1 and 9.3.3-2). Some of the other results, though more technical, are also important in the next section, where we prove the contractiveness of object types.

The casual reader may wish to look only at Propositions 9.3.3-8 and 9.3.3-9 and Lemmas 9.3.3-1 and 9.3.3-2; none of the intermediate results is particularly surprising.

We use some further notation for binary relations and sets of binary relations:

- X°: The set of finite elements of X.
- $\mathcal{T}(X)$: The transitive closure of X.
- $\mathcal{U}(X)$: The chain completion of X: $(x,y) \in \mathcal{U}(X)$ if and only if either $(x,y) = (\perp,\perp)$ or $(x,y) = (\sqcup_i x_i, \sqcup_i y_i)$ for $\langle (x_i, y_i) \rangle$ an increasing sequence in X.
- **UR**: The set of uniform binary relations.
- **CR**: The set of complete binary relations.
- **NUSR**: The set of nonempty, uniform, symmetric binary relations.
- **NUPER**: The set of nonempty, uniform pers.

Proposition 9.3.3-1

If $X_i \in$ **NUSR** for all $i \in I$ then $\cup_{i \in I} X_i \in$ **NUSR**.

Proof

Nonemptiness, uniformity, and symmetry are all preserved by arbitrary unions.

□

Proposition 9.3.3-2

If $X \in$ **UR** then $\mathcal{T}(X) \in$ **UR**.

Proof

If $(x,y) \in \mathcal{T}(X)$ then $x = z_0 \; X \; z_1 \; ... \; z_{n-1} \; X \; z_n = y$ for some $z_0, ..., z_n$, hence $p_i(x) = p_i(z_0) \; X \; p_i(z_1) \; ... \; p_i(z_{n-1}) \; X \; p_i(z_n) = p_i(y)$ for all i, by uniformity, so $(p_i(x),p_i(y)) \in \mathcal{T}(X)$.

□

Proposition 9.3.3-3

If $X \in$ **NUSR** then $\mathcal{T}(X) \in$ **NUPER**.

Proof

We check that $\mathcal{T}(X) \in$ **NUPER**, and that in addition $\mathcal{T}(X)$ is transitive.

- Transitive closure preserves nonemptiness.
- Transitive closure preserves symmetry.
- Transitive closure preserves uniformity, by Proposition 9.3.3-2.
- Transitive closure establishes transitivity.

□

Proposition 9.3.3-4

If (x,y) is finite and (x,y) $\in \mathcal{U}(X)$ then (x,y) $\in$ X or (x,y) = ($\perp$,$\perp$).

Proof

Assume (x,y) $\in \mathcal{U}(X)$ and (x,y) $\neq$ ($\perp$, $\perp$). Let $x = \sqcup_k x_k$ and $y = \sqcup_k y_k$, where $\langle(x_k, y_k)\rangle$ is an increasing sequence in X. Since (x,y) is finite, there exists k such that $(x,y) = (x_k,y_k)$.

□

Proposition 9.3.3-5

If X $\in$ **UR** then $\mathcal{U}(X) \in$ **UR**.

Proof

Consider (x,y) $\in \mathcal{U}(X)$. We wish to show that $(p_i(x),p_i(y)) \in \mathcal{U}(X)$ for all i. In case (x,y) = ($\perp$,$\perp$), this is trivial, as ($p_i(x)$,$p_i(y)$) = ($\perp$,$\perp$). Otherwise, let $x = \sqcup_k x_k$ and $y = \sqcup_k y_k$, where $\langle(x_k, y_k)\rangle$ is an increasing sequence in X. Clearly $(p_i(x), p_i(y)) \sqsubseteq (\sqcup_k x_k, \sqcup_k y_k)$, and since $(p_i(x), p_i(y))$ is finite there exists a j such that $(p_i(x), p_i(y)) \sqsubseteq (x_j, y_j)$. By monotonicity, $(p_i(x), p_i(y)) \sqsubseteq (p_i(x_j), p_i(y_j))$. On the other hand, $(x_j, y_j) \sqsubseteq (x,y)$, and hence $(p_i(x_j), p_i(y_j)) \sqsubseteq (p_i(x), p_i(y))$. Hence for all i there exists j such that $p_i(x) = p_i(x_j)$ and $p_i(y) = p_i(y_j)$. Since X is uniform, it follows that $(p_i(x),p_i(y)) \in$ X for all i, and hence that $(p_i(x),p_i(y)) \in \mathcal{U}(X)$ for all i.

□

Proposition 9.3.3-6

If X $\in$ **UR** then $\mathcal{U}(X) \in$ **CR**.

Proof

Clearly, ($\perp$,$\perp$) $\in \mathcal{U}(X)$.

In addition, consider an increasing chain in $\mathcal{U}(X)$, $\langle(x_i, y_i)\rangle$, with limit (x,y). We show that (x,y) $\in \mathcal{U}(X)$ by finding an increasing chain in X, $\langle(x_i', y_i')\rangle$ with limit (x,y).

The case of X empty is trivial.

In case X is nonempty, let $(x_i', y_i') = (p_i(x_i), p_i(y_i))$. If $\langle(x_i, y_i)\rangle$ is increasing, then so is $\langle(p_i(x_i), p_i(y_i))\rangle$. Since chain completion preserves uniformity by Proposition 9.3.3-5, $(x_i', y_i') \in \mathcal{U}(X)$. Moreover, by Proposition 9.3.3-4, $\langle(x_i', y_i')\rangle$ is an increasing sequence in X (since ($\perp$,$\perp$) $\in$ X as X is nonempty).

Finally we check that the limit of this constructed sequence is (x,y). Clearly, $(\sqcup_i x_i', \sqcup_i y_i') \sqsubseteq (x,y)$, since $(x_i', y_i') \sqsubseteq (x_i,y_i)$ for all i and $(x,y) = (\sqcup_i x_i, \sqcup_i y_i)$. To show that $(\sqcup_i x_i', \sqcup_i y_i') \sqsupseteq (x,y)$, we prove that for all j there exists i such that $(p_j(x),p_j(y)) \sqsubseteq (x_i', y_i')$. By finiteness considerations, for all j there exists k such that $(p_j(x),p_j(y)) \sqsubseteq (x_k, y_k)$. Letting i = max(j,k), we obtain $(p_j(x),p_j(y)) \sqsubseteq (x_i, y_i)$, and then $(p_j(x),p_j(y)) \sqsubseteq (p_i(x_i), p_i(y_i)) = (x_i', y_i')$, the desired result.

□

Proposition 9.3.3-7

If X $\in$ **NUPER** then $\mathcal{U}(X) \in$ **CUPER**.

Proof

We check that $\mathcal{U}(X) \in$ **NUPER**, and that in addition $\mathcal{U}(X)$ is complete.

- Chain completion preserves nonemptiness.
- Chain completion preserves symmetry.
- Chain completion preserves uniformity, by Proposition 9.3.3-5.

- Chain completion preserves transitivity in presence of uniformity: If $(x,y), (y,z) \in \mathcal{U}(X)$, then $(p_i(x),p_i(y)), (p_i(y),p_i(z)) \in \mathcal{U}(X)$ for all i, by uniformity, and $(p_i(x),p_i(y)), (p_i(y),p_i(z)) \in X$ for all i by Proposition 9.3.3-4. By transitivity of X, $(p_i(x),p_i(z)) \in X$ for all i, and hence $(x,z) \in \mathcal{U}(X)$.
- Chain completion establishes completeness in presence of uniformity, by Proposition 9.3.3-6.

□

Proposition 9.3.3-8

Let $X \in$ **NUSR**. Then $\mathcal{C}(X) = \mathcal{U}(\mathcal{T}(X^\circ))$.

Proof

Let $Y = X^\circ$. Note that $Y \in$ **NUSR**.

First, $X \subseteq \mathcal{U}(\mathcal{T}(Y))$, and in fact $X \subseteq \mathcal{U}(Y)$: If $(x,y) \in X$ then $(p_i(x),p_i(y)) \in X$ for all i, by uniformity, and hence $(p_i(x),p_i(y)) \in Y$ for all i; therefore, $(x,y) \in \mathcal{U}(Y)$.

Next, $\mathcal{U}(\mathcal{T}(Y)) \in$ **CUPER**, by Propositions 9.3.3-3 and 9.3.3-7.

Finally, consider any other **CUPER** R such that $X \subseteq R$. We show that $\mathcal{U}(\mathcal{T}(Y)) \subseteq R$. It is obvious that $\mathcal{U}(\mathcal{T}(X)) \subseteq R$. So it suffices to remark that $\mathcal{U}(\mathcal{T}(X)) \supseteq \mathcal{U}(\mathcal{T}(Y))$.

□

Proposition 9.3.3-9

Let $X \in$ **NUSR**, and let (x,y) be finite. If $(x,y) \in \mathcal{C}(X)$ then there exist $z_0, ..., z_n$ such that $x = z_0$ X z_1 ... z_{n-1} X $z_n = y$. In fact, each of $z_0, ..., z_n$ can be chosen finite and of rank no greater than that of x and y.

Proof

First recall that $\mathcal{C}(X) = \mathcal{U}(\mathcal{T}(Y))$, where Y is the finite part of X, by Proposition 9.3.3-8. If $(x,y) \in \mathcal{C}(X)$ then $(x,y) \in \mathcal{T}(Y)$, by Proposition 9.3.3-4 (since clearly $(\perp,\perp) \in Y$). So there exist $z_0, ..., z_n$ such that $x = z_0$ Y z_1 ... z_{n-1} Y $z_n = y$. By uniformity of Y, if x and y have rank at most k, then $x = p_k(z_0)$ Y $p_k(z_1)$ ... $p_k(z_{n-1})$ Y $p_k(z_n) = y$, and hence $x = p_k(z_0)$ X $p_k(z_1)$ ... $p_k(z_{n-1})$ X $p_k(z_n) = y$.

□

Proposition 9.3.3-10

If
- $X, Y \in$ **NUSR**,
- f is a continuous function,
- $(x,y) \in X$ implies $(f(x), f(y)) \in Y$

then $(x,y) \in \mathcal{C}(X)$ implies $(f(x), f(y)) \in \mathcal{C}(Y)$.

□

Lemma 9.3.3-1

If
- $R_i \in$ **CUPER** for all $i \in I$, and $S \in$ **CUPER**,
- f is a continuous function,
- for all i, $(x,y) \in R_i$ implies $(f(x), f(y)) \in S$,

then $(x,y) \in \sqcup_{i \in I} R_i$ implies $(f(x), f(y)) \in S$.

□

Proposition 9.3.3-10 and Lemma 9.3.3-1 are special cases of the following results:

Proposition 9.3.3-11

If

- X, Y ∈ **NUSR**,
- f, g are continuous functions,
- (x,y) ∈ X implies (f(x), f(y)) ∈ Y and (f(x), g(y)) ∈ Y,

then (x,y) ∈ $\mathcal{C}$(X) implies (f(x), g(y)) ∈ $\mathcal{C}$(Y).

Proof

Assume (x,y) is finite. Then, by Proposition 9.3.3-9, (x,y) ∈ $\mathcal{C}$(X) implies that for some z_0, ..., z_n, we have $x = z_0$ X z_1 ... z_{n-1} X $z_n = y$. By hypothesis, $f(x) = f(z_0)$ Y $f(z_1)$ ... $f(z_{n-1})$ Y $g(z_n) = g(y)$. Since Y ⊆ $\mathcal{C}$(Y) and since $\mathcal{C}$(Y) is transitive, it follows that (f(x),g(y)) ∈ $\mathcal{C}$(Y).

Now consider (x,y) not necessarily finite. If (x,y) ∈ $\mathcal{C}$(X), then for all i, $(p_i(x),p_i(y))$ ∈ $\mathcal{C}$(X), by uniformity. Now the argument for finite elements yields $(f(p_i(x)), g(p_i(y)))$ ∈ $\mathcal{C}$(Y). Since f and g are continuous, $(f(x), g(y)) = (\sqcup_i f(p_i(x)), \sqcup_i g(p_i(y)))$. By completeness it follows that (f(x), g(y)) ∈ $\mathcal{C}$(Y).

□

Lemma 9.3.3-2

If

- R_i ∈ **CUPER** for all i ∈ I, and S ∈ **CUPER**,
- f and g are continuous functions,
- for all i, (x,y) ∈ R_i implies (f(x), f(y)) ∈ S and (f(x), g(y)) ∈ S,

then (x,y) ∈ $\sqcup_{i\in I} R_i$ implies (f(x), g(y)) ∈ S.

Proof

Let X = $\cup_{i\in I} R_i$. By Proposition 9.3.3-1, X ∈ **NUSR**. Let Y = S. The hypotheses say that (x,y) ∈ X implies (f(x), f(y)) ∈ Y and (f(x), g(y)) ∈ Y. Applying Proposition 9.3.3-11, we obtain (x,y) ∈ $\mathcal{C}$(X) implies (f(x), g(y)) ∈ $\mathcal{C}$(Y), that is, (x,y) ∈ $\sqcup_{i\in I} R_i$ implies (f(x), g(y)) ∈ S.

□

9.3.4 Metric Properties

We start by checking that the object-type construction is contractive.

Proposition 9.3.4-1

For all n, if R_i ∈ **CUPER** for all i ∈ I, S_j ∈ **CUPER** for j ∈ J, and for all i ∈ I there exists j ∈ J such that $p_n(R_i) \subseteq p_n(S_j)$, then $p_n(\sqcup_{i\in I} R_i) \subseteq p_n(\sqcup_{j\in J} S_j)$.

Proof

Assume (v,v') ∈ $p_n(\sqcup_{i\in I} R_i)$, that is, v,v' have rank at most n and (v,v') ∈ $\mathcal{C}(\cup_{i\in I} R_i)$. By Proposition 9.3.3-5, $\cup_{i\in I} R_i$ ∈ **NUSR**. By Proposition 9.3.3-9, we obtain (v,v') ∈ $\mathcal{T}(\cup_{i\in I} p_n(R_i))$. Since for all i ∈ I there exists j ∈ J such that $p_n(R_i) \subseteq p_n(S_j)$, we also obtain (v,v') ∈ $\mathcal{T}(\cup_{j\in J} p_n(S_j))$. Hence, (v,v') ∈ $p_n(\sqcup_{j\in J} S_j)$.

□

Proposition 9.3.4-2

If $T_i(R_1,...,R_k)$ is nonexpansive in R_1, ..., R_k for all i ∈ I then $[[m_i:T_i(R_1,...,R_k)\ ^{i\in I}]]$ is contractive in R_1, ..., R_k.

Proof

We give the proof for k=1; the general case is similar to this one.

Let R and R' be equal up to rank n (at least).

Consider $F \in \mathbf{Gen}$ such that $F \preccurlyeq \lambda(S)\langle\langle m_i : S \to T_i(R)^{\ i\in I}\rangle\rangle$. If $F = \lambda(S)\langle\langle m_i : S \to T_i(R)^{\ i\in I}, m_j : S \to T_j^{\ j\in J}\rangle\rangle$, let $F' = \lambda(S)\langle\langle m_i : S \to T_i(R')^{\ i\in I}, m_j : S \to T_j^{\ j\in J}\rangle\rangle$. Now, $F' \in \mathbf{Gen}$ and $F' \preccurlyeq \lambda(S)\langle\langle m_i : S \to T_i(R')^{\ i\in I}\rangle\rangle$. Furthermore, we obtain that F and F' are equal up to rank n+1, since each T_i is nonexpansive, $\to$ is contractive, $\sqcap$ is nonexpansive (using the formula for $\lambda(S)\langle\langle ...\rangle\rangle$ in terms of $\to$ and $\sqcap$). In short, for every extension F of $\lambda(S)\langle\langle m_i : S \to T_i(R)^{\ i\in I}\rangle\rangle$ there exists an extension F' of $\lambda(S)\langle\langle m_i : S \to T_i(R')^{\ i\in I}\rangle\rangle$ equal to F up to rank n+1.

Then $\mu(S)F(S)$ and $\mu(S)F'(S)$ coincide up to rank n+1, and hence $[[m_i : T_i(R)^{\ i\in I}]]$ is included in $[[m_i : T_i(R')^{\ i\in I}]]$ up to rank n+1, by Proposition 9.3.4-1. We obtain the converse analogously. Hence, $[[m_i : T_i(R)^{\ i\in I}]]$ and $[[m_i : T_i(R')^{\ i\in I}]]$ coincide up to rank n+1.

□

With the machinery now available, it is simple to check that union and intersection are nonexpansive, and that they preserve contractiveness:

Proposition 9.3.4-3

If $T(R_1,...,R_{k+1})$ is contractive (nonexpansive) in R_1, ..., R_{k+1} and $S(R_1,...,R_k)$ is nonexpansive in R_1, ..., R_k, then $\sqcup_{R_{k+1}\in\mathbf{CUPER},\ R_{k+1}\subseteq S(R_1,...,R_k)}\ T(R_1,...,R_{k+1})$ is contractive (nonexpansive) in R_1, ..., R_k.

Proof

Again, we give the proof for k=1; we prove only the contractiveness claim, as the other one is similar.

Suppose that R and R' are equal up to rank n. We want to show that $\sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R)}\ T(R,R_2)$ and $\sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$ are equal up to rank n+1. Let $(v,v') \in \sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R)}\ T(R,R_2)$ of rank n+1. We wish to show that $(v,v') \in \sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$. By Lemma 9.3.3-1, it suffices to prove that if $(v,v') \in T(R,R_2)$ for some $R_2 \subseteq S(R)$ then $(v,v') \in \sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$. So assume that $(v,v') \in T(R,R_2)$ and $R_2 \subseteq S(R)$. Since T is contractive in R_2 and (v,v') are of rank n+1, we have $(v,v') \in T(R,p_n(R_2))$. Similarly, since T is contractive in R, and R and R' are equal up to rank n, we have $(v,v') \in T(R',p_n(R_2))$. In addition, $p_n(R_2) \in \mathbf{CUPER}$ and $p_n(R_2) \subseteq S(R)$. Since S is nonexpansive in R, and R and R' are equal up to rank n, we have also $p_n(R_2) \subseteq S(R')$. It follows that $(v,v') \in \sqcup_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$.

□

Proposition 9.3.4-4

If $T(R_1,...,R_{k+1})$ is contractive (nonexpansive) in R_1, ..., R_{k+1} and $S(R_1,...,R_k)$ is nonexpansive in R_1, ..., R_k, then $\sqcap_{R_{k+1}\in\mathbf{CUPER},\ R_{k+1}\subseteq S(R_1,...,R_k)}\ T(R_1,...,R_{k+1})$ is contractive (nonexpansive) in R_1, ..., R_k.

Proof

We give the contractiveness proof for k=1.

Suppose that R and R' are equal up to rank n. We want to show that $\sqcap_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R)}\ T(R,R_2)$ and $\sqcap_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$ are equal up to rank n+1. Let $(v,v') \in \sqcap_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R)}\ T(R,R_2)$ of rank n+1. Pick an R_2 such that $R_2 \in \mathbf{CUPER}$ and $R_2 \subseteq S(R')$, to show that $(v,v') \in T(R',R_2)$. It will follow that $(v,v') \in \sqcap_{R_2\in\mathbf{CUPER},\ R_2\subseteq S(R')}\ T(R',R_2)$. Since R and R' agree up to rank n, and since S is nonexpansive, $R_2 \subseteq S(R')$ implies $p_n(R_2) \subseteq S(R)$. In addition, $p_n(R_2) \in \mathbf{CUPER}$. Hence $(v,v') \in T(R,p_n(R_2))$. Since T is contractive in R, it follows that $(v,v') \in T(R',p_n(R_2))$. Since T is contractive in R_2, it follows that $(v,v') \in T(R',R_2)$.

□

9.3.5 Semantic Properties of Objects

In this section we prove some fundamental properties of records, objects, and their types. These properties serve as a convenient basis for validating the rules of $\mathbf{FOb}_{<:\mu}$, in the next section.

The first proposition says that the record-type construction is monotonic.

Proposition 9.3.5-1

If $T_i \subseteq T_i'$ then $\langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle \subseteq \langle\langle m_i : T_i' \ ^{i\in I}\rangle\rangle$.

Proof

Directly from the definitions.

□

The next proposition enables us to equate two explicitly given records, possibly of different lengths.

Proposition 9.3.5-2

Let $I \cap K = \emptyset$. If $(v_i,v_i') \in T_i$ for all $i \in I$ then $(\langle\langle m_i = v_i \ ^{i\in I}\rangle\rangle, \langle\langle m_i = v_i' \ ^{i\in I}, m_k=u_k' \ ^{k\in K}\rangle\rangle) \in \langle\langle m_i : T_i \ ^{i\in I}\rangle\rangle$.

Proof

Directly from the definitions.

□

The final proposition about records concerns updates.

Proposition 9.3.5-3

If $(v,v') \in \langle\langle m_i:T_i \ ^{i\in I}\rangle\rangle$ and $(u,u')\in T_k$ for some $k \in I$ then $(v\langle m_k\leftarrow u\rangle, v'\langle m_k\leftarrow u'\rangle) \in \langle\langle m_i:T_i \ ^{i\in I}\rangle\rangle$.

Proof

Directly from the definitions. Note however the special case $(v,v') =(\bot,\bot)$, where $v\langle m_k\leftarrow u\rangle$ and $v'\langle m_k\leftarrow u'\rangle$ are both $\bot$.

□

We have analogous propositions for objects. The first proposition means that the desired subtypings hold with our semantics.

Proposition 9.3.5-4

If $I \subseteq J$ then $[[m_j : T_j \ ^{j\in J}]] \subseteq [[m_i : T_i \ ^{i\in I}]]$.

Proof

All extensions of $\lambda(S)\langle\langle m_j : S\rightarrow T_j \ ^{j\in J}\rangle\rangle$ are also extensions of $\lambda(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$, hence the result.

□

The next group of results concerns the equality of explicitly given objects and the semantic properties of method invocation and override.

Proposition 9.3.5-5

Let $I \cap K = \emptyset$. If $(v_j,v_j') \in [[m_i : T_i \ ^{i\in I}]]\rightarrow T_j$ for all $j \in I$, then $(\langle\langle m_i=v_i \ ^{i\in I}\rangle\rangle, \langle\langle m_i=v_i' \ ^{i\in I}, m_k=u_k' \ ^{k\in K}\rangle\rangle) \in \mu(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$, and hence $(\langle\langle m_i=v_i \ ^{i\in I}\rangle\rangle, \langle\langle m_i=v_i' \ ^{i\in I}, m_k=u_k' \ ^{k\in K}\rangle\rangle) \in [[m_i : T_i \ ^{i\in I}]]$.

Proof

Let $T = [[m_i : T_i \ ^{i\in I}]]$ and let $R = \mu(S)\langle\langle m_i : S\rightarrow T_i \ ^{i\in I}\rangle\rangle$. By unfolding, it suffices to check that $(\langle\langle m_i=v_i \ ^{i\in I}\rangle\rangle, \langle\langle m_i=v_i' \ ^{i\in I}, m_k=u_k' \ ^{k\in K}\rangle\rangle) \in \langle\langle m_i : R\rightarrow T_i \ ^{i\in I}\rangle\rangle$. The hypothesis and Proposition 9.3.5-2

yield that $(\langle\langle m_i{=}v_i\ ^{i\in I}\rangle\rangle, \langle\langle m_i{=}v_i'\ ^{i\in I}\rangle\rangle) \in \langle\langle m_i : T{\to}T_i\ ^{i\in I}\rangle\rangle$. The result follows since $R \subseteq T$, $\to$ is anti-monotonic, and $\langle\langle ... \rangle\rangle$ is monotonic by Proposition 9.3.5-1.

□

Proposition 9.3.5-6

If $(v,v') \in [[m_i : T_i\ ^{i\in I}]]$ then either $v = v' = \bot$ or $v, v' \in (L{\to}D)$.

Proof

The function $f(x) = x$ is continuous, and

$$[[m_i{:}T_i\ ^{i\in I}]] = \sqcup \{ \mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)\langle\langle m_i{:}S{\to}T_i\ ^{i\in I}\rangle\rangle\}$$

Lemma 9.3.3-1 applies. It suffices to prove that if $(v,v') \in \mu(S)F(S)$ and $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$ then $(v, v') \in \{(\bot,\bot)\} \cup (L{\to}D) \times (L{\to}D)$.
If $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$, then $F(S)$ has the form $\langle\langle m_j : S{\to}T_j\ ^{j\in J}\rangle\rangle$. If $(v,v') \in \mu(S)F(S)$, unfolding yields $(v,v') \in \langle\langle m_j : (\mu(S)F(S)){\to}T_j\ ^{j\in J}\rangle\rangle$. By definition of $\langle\langle ... \rangle\rangle$, we have $(v, v') \in \{(\bot,\bot)\} \cup (L{\to}D) \times (L{\to}D)$.

□

Proposition 9.3.5-7

If $(v,v') \in [[m_i : T_i\ ^{i\in I}]]$ then $v(m_k), v'(m_k) \in (D{\to}D)$ for all $k \in I$.

Proof

The function $f(x) = x(m_k)$ is continuous, and again Lemma 9.3.3-1 applies. It suffices to prove that if $(v,v') \in \mu(S)F(S)$, $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$, and $k \in I$, then $(v(m_k), v'(m_k)) \in (D{\to}D) \times (D{\to}D)$.
If $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$, then $F(S)$ has the form $\langle\langle m_k : S{\to}T_k, ...\rangle\rangle$. If $(v,v') \in \mu(S)F(S)$, unfolding yields $(v,v') \in \langle\langle m_j : (\mu(S)F(S)){\to}T_j, ...\rangle\rangle$. By Proposition 9.3.5-6, either $(v, v')=(\bot,\bot)$ or $v, v' \in (L{\to}D)$. If $(v, v')=(\bot,\bot)$, then the result is trivial. If $v, v' \in (L{\to}D)$, we obtain $(v(m_k), v'(m_k)) \in (\mu(S)F(S)){\to}T_j$, and hence $(v(m_k), v'(m_k)) \in (D{\to}D) \times (D{\to}D)$.

□

Proposition 9.3.5-8

If $(v,v') \in [[m_i : T_i\ ^{i\in I}]]$ then $(v(m_k)(v), v'(m_k)(v')) \in T_k$ for all $k \in I$.

Proof

The function $f(x) = x(m_k)(x)$ is continuous, and again Lemma 9.3.3-1 applies. It suffices to prove that if $(v,v') \in \mu(S)F(S)$, $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$, and $k \in I$, then $(v(m_k)(v), v'(m_k)(v')) \in T_k$.
If $F \preccurlyeq \lambda(S)\langle\langle m_i : S{\to}T_i\ ^{i\in I}\rangle\rangle$, then $F(S)$ has the form $\langle\langle m_k : S{\to}T_k, ...\rangle\rangle$. If $(v,v') \in \mu(S)F(S)$, unfolding yields $(v,v') \in \langle\langle m_k : (\mu(S)F(S)){\to}T_k, ...\rangle\rangle$. By Proposition 9.3.5-6, either $(v, v')=(\bot,\bot)$ or $v, v' \in (L{\to}D)$. If $(v,v') = (\bot,\bot)$ then the result is trivial. So we assume that $v, v' \in (L{\to}D)$. An application yields $(v(m_k),v'(m_k)) \in (\mu(S)F(S)){\to}T_k$, and then a second application yields the desired result, $(v(m_k)(v),v'(m_k)(v')) \in T_k$.

□

Proposition 9.3.5-9

If $(v,v') \in [[m_i : T_i\ ^{i\in I}]]$ and $(u,u') \in [[m_i : T_i\ ^{i\in I}]]{\to}T_k$ for some $k \in I$, then $(v\langle m_k{\leftarrow}u\rangle, v'\langle m_k{\leftarrow}u'\rangle) \in [[m_i : T_i\ ^{i\in I}]]$.

Proof

The functions $f(x) = x\langle m_k \leftarrow u\rangle$ and $g(x) = x\langle m_k \leftarrow u'\rangle$ are continuous in x, and now Lemma 9.3.3-2 applies. It suffices to prove that if $(v,v') \in \mu(S)F(S)$, $F \preccurlyeq \lambda(S)\langle\langle m_i : S \to T_i\ ^{i\in I}\rangle\rangle$, $k \in I$, and $(u,u') \in [[m_i : T_i\ ^{i\in I}]] \to T_k$, then $(v\langle m_k \leftarrow u\rangle,\ v'\langle m_k \leftarrow u\rangle) \in [[m_i : T_i\ ^{i\in I}]]$ and $(v\langle m_k \leftarrow u\rangle,\ v'\langle m_k \leftarrow u'\rangle) \in [[m_i : T_i\ ^{i\in I}]]$.

If $F \preccurlyeq \lambda(S)\langle\langle m_i : S \to T_i\ ^{i\in I}\rangle\rangle$, then F(S) has the form $\langle\langle m_k : S \to T_k, ...\rangle\rangle$. If $(v,v') \in \mu(S)F(S)$, unfolding yields $(v,v') \in \langle\langle m_k : (\mu(S)F(S)) \to T_k, ...\rangle\rangle$. By Proposition 9.3.5-6, either $(v, v') = (\bot,\bot)$ or $v, v' \in (L \to D)$. If $(v,v') = (\bot,\bot)$ then the result is trivial. So we assume $v, v' \in (L \to D)$. Now, $(u,u') \in [[m_i : T_i\ ^{i\in I}]] \to T_k$ implies $(u,u') \in (\mu(S)F(S)) \to T_k$. So by Proposition 9.3.5-3 $(v\langle m_k \leftarrow u\rangle, v'\langle m_k \leftarrow u'\rangle) \in \langle\langle m_i : (\mu(S)F(S)) \to T_i\ ^{i\in I}\rangle\rangle$. By folding, $(v\langle m_k \leftarrow u\rangle, v'\langle m_k \leftarrow u'\rangle) \in \mu(S)F(S) \subseteq [[m_i : T_i\ ^{i\in I}]]$.

Considering the case u = u', we obtain also $(v\langle m_k \leftarrow u\rangle, v'\langle m_k \leftarrow u\rangle) \in [[m_i : T_i\ ^{i\in I}]]$.

□

9.4 The Interpretation of Types and Typed Terms

Sections 9.4.1 and 9.4.2 give the interpretation of terms and types. Section 9.4.3 checks the soundness of our rules under this interpretation.

9.4.1 Interpreting Typed Terms

The semantics of a typed term is the semantics of its erasure: if a is a typed term then $[\![a]\!]_\rho = [\![e(a)]\!]_\rho$. The erasure function is a translation from the typed terms of $\mathbf{FOb}_{<:\mu}$ to the untyped calculus:

$$
\begin{aligned}
e(x) &= x \\
e(\lambda(x{:}A)b) &= \lambda(x)e(b) \\
e(b(a)) &= e(b)(e(a)) \\
e(\lambda(X{<:}A)b) &= e(b) \\
e(b(A)) &= e(b) \\
e(\text{pack } X{<:}A{=}C,\ b\{X\}{:}B\{X\}) &= e(b) \\
e(\text{open } c \text{ as } X{<:}A, x{:}B \text{ in } d{:}D) &= e(d\{x \leftarrow c\}) \\
e([m_i = \varsigma(x_i{:}A)c_i\ ^{i\in 1..n}]) &= [m_i = \varsigma(x_i)e(c_i)\ ^{i\in 1..n}] \\
e(a.m) &= e(a).m \\
e(a.m \Leftarrow \varsigma(x{:}A)c) &= e(a).m \Leftarrow \varsigma(x)e(c)
\end{aligned}
$$

In this definition, we ignore the primitives fold, unfold, and value-level recursion, because they are somewhat superfluous. Recursive values can be obtained with the definable fixpoint combinator **Y**; and since we provide exact solutions for recursive-type equations, fold and unfold can both be interpreted as the identity function.

9.4.2 Interpreting Types

In this section, we define the semantics function for types:

$$[\![\ ]\!] : (TV \to \mathbf{CUPER}) \to (TE \to \mathbf{CUPER})$$

where TV is the set of type variables and TE the set of type expressions. A mapping η in $TV \to \mathbf{CUPER}$ is a type environment, and we write $[\![A]\!]_\eta$ for the semantics of a type A with the environment η.

We set:

$$
\begin{aligned}
[\![X]\!]_\eta &= \eta(X) \\
[\![A\to B]\!]_\eta &= [\![A]\!]_\eta \to [\![B]\!]_\eta \\
[\![\forall(X<:B)A]\!]_\eta &= \sqcap_{R\in\mathbf{CUPER},\, R\sqsubseteq[\![B]\!]_\eta} [\![A]\!]_{\eta\langle X\leftarrow R\rangle} \\
[\![\exists(X<:B)A]\!]_\eta &= \sqcup_{R\in\mathbf{CUPER},\, R\sqsubseteq[\![B]\!]_\eta} [\![A]\!]_{\eta\langle X\leftarrow R\rangle} \\
[\![\mu(X)A]\!]_\eta &= \mu(T)\, [\![A]\!]_{\eta\langle X\leftarrow T\rangle} \\
[\![[m_i{:}C_i\ ^{i\in 1..n}]]\!]_\eta &= [(m_i : [\![C_i]\!]_\eta\ ^{i\in 1..n})] \\
[\![\mathrm{Top}]\!]_\eta &= (D - \{*\}) \times (D - \{*\})
\end{aligned}
$$

The definition for recursive types is proper because of the connection between contractiveness and formal contractiveness:

Proposition 9.4.2-1

If A is a well-formed type expression then $[\![A]\!]_{\eta\langle X\leftarrow R\rangle}$ is nonexpansive in R. If A is formally contractive in X then $[\![A]\!]_{\eta\langle X\leftarrow R\rangle}$ is contractive in R.

Proof

Let $\eta' = \eta\langle X\leftarrow R\rangle$. The proof is a joint induction. The cases of type variables, $\to$, and μ are well known. The case of Top is trivial. Propositions 9.3.4-3 and 9.3.4-4 deal with bounded quantification. Proposition 9.3.4-2 deals with object types.

□

It follows that every well-formed type expression denotes a nonexpansive function on **CUPER** (as a function of its free variables).

9.4.3 Verification of Selected Rules

We say that E and η are consistent in the usual sense: if X<:A appears in E then $\eta(X) \subseteq [\![A]\!]_\eta$. We say that E, η, and (ρ,ρ') are consistent when, in addition, if x:A appears in E then $(\rho(x),\rho'(x)) \in [\![A]\!]_\eta$. Next we check that if η and (ρ,ρ') are consistent with E then $E \vdash B<:A$ implies $[\![B]\!]_\eta \subseteq [\![A]\!]_\eta$, $E \vdash a : A$ implies $([\![a]\!]_\rho, [\![a]\!]_{\rho'}) \in [\![A]\!]_\eta$, and $E \vdash a \leftrightarrow a' : A$ implies $([\![a]\!]_\rho, [\![a']\!]_{\rho'}) \in [\![A]\!]_\eta$. An immediate corollary of these verifications is that no well-typed term has $*$ as its denotation.

The proof is by induction on derivations. We start with some basic rules from the λ-calculus, because our induction is superficially different to those of Amadio, Cardone, and others. We focus on the crucial rules for objects. The arguments for the remaining rules are either well known, or special cases of the ones given.

1. For (Eq Fun):

$$\frac{E, x{:}A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x{:}A)b \ \leftrightarrow\ \lambda(x{:}A)b' : A\to B}$$

Consider η and (ρ,ρ') consistent with E, and $(v,v') \in [\![A]\!]_\eta$. Let $\theta = \rho\langle x\leftarrow v\rangle$ and $\theta' = \rho'\langle x\leftarrow v'\rangle$. We have that $([\![\lambda(x{:}A)b]\!]_\rho)v = (\lambda(u)[\![b]\!]_{\rho\langle x\leftarrow u\rangle})(v)$, and hence $[\![\lambda(x{:}A)b]\!]_\rho(v) = [\![b]\!]_\theta$. Similarly, $[\![\lambda(x{:}A)b']\!]_{\rho'}(v') = [\![b']\!]_{\theta'}$. Since η and (θ,θ') are consistent with E, x:A, the hypothesis applies, yielding $([\![b]\!]_\theta, [\![b']\!]_{\theta'}) \in [\![B]\!]_\eta$, hence $([\![\lambda(x{:}A)b]\!]_\rho(v), [\![\lambda(x{:}A)b']\!]_{\rho'}(v')) \in [\![B]\!]_\eta$, and

$$([\![\lambda(x{:}A)b]\!]_\rho, [\![\lambda(x{:}A)b']\!]_{\rho'}) \in [\![A]\!]_\eta \to [\![B]\!]_\eta$$

that is,

$$([\![\lambda(x{:}A)b]\!]_\rho, [\![\lambda(x{:}A)b']\!]_{\rho'}) \in [\![A\to B]\!]_\eta$$

2. For (Eq Appl):

$$\frac{E \vdash b \leftrightarrow b' : A \to B \qquad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \ \leftrightarrow \ b'(a') : B}$$

The first hypothesis yields $([\![b]\!]_\rho, [\![b']\!]_{\rho'}) \in [\![A \to B]\!]_\eta$, so $([\![b]\!]_\rho, [\![b']\!]_{\rho'}) \in [\![A]\!]_\eta \to [\![B]\!]_\eta$. Hence, $[\![b]\!]_\rho$, $[\![b']\!]_{\rho'} \in (D \to D)$. We obtain $([\![a]\!]_\rho, [\![a']\!]_{\rho'}) \in [\![A]\!]_\eta$ from the second hypothesis. Hence, $([\![b]\!]_\rho([\![a]\!]_\rho), [\![b']\!]_{\rho'}([\![a']\!]_{\rho'})) \in [\![B]\!]_\eta$, with $[\![b(a)]\!]_\rho = [\![b]\!]_\rho([\![a]\!]_\rho)$ and $[\![b'(a')]\!]_{\rho'} = [\![b']\!]_{\rho'}([\![a']\!]_{\rho'})$. Finally, we derive

$$([\![b(a)]\!]_\rho, [\![b'(a')]\!]_{\rho'}) \in [\![B]\!]_\eta$$

3. For (Eq Fun2<:):

$$\frac{E, X<:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X<:A)b \ \leftrightarrow \ \lambda(X<:A)b' : \forall(X<:A)B}$$

Consider η and (ρ, ρ') consistent with E, and $R \in$ **CUPER** such that $R \subseteq [\![A]\!]_\eta$. Let $\vartheta = \eta\langle X \leftarrow R\rangle$. We have that $[\![\lambda(X<:A)b]\!]_\rho = [\![b]\!]_\rho$. Similarly, $[\![\lambda(X<:A)b']\!]_{\rho'} = [\![b']\!]_{\rho'}$. Since ϑ and (ρ, ρ') are consistent with E, X<:A, the hypothesis applies, yielding $([\![b]\!]_\rho, [\![b']\!]_{\rho'}) \in [\![B]\!]_\vartheta$, hence $([\![\lambda(X<:A)b]\!]_\rho, [\![\lambda(X<:A)b']\!]_{\rho'}) \in [\![B]\!]_\vartheta$. Since this is true independently of the choice of R,

$$([\![\lambda(X<:A)b]\!]_\rho, [\![\lambda(X<:A)b']\!]_{\rho'}) \in [\![\forall(X<:A)B]\!]_\eta$$

4. For (Eq Appl2<:):

$$\frac{E \vdash b \leftrightarrow b' : \forall(X<:A)B \qquad E \vdash C <: A}{E \vdash b(C) \ \leftrightarrow \ b'(C) : B\{X \leftarrow C\}}$$

Since $[\![C]\!]_\eta \subseteq [\![A]\!]_\eta$, we have that $[\![\forall(X<:A)B]\!]_\eta \subseteq [\![B]\!]_{\eta\langle X \leftarrow [\![C]\!]\eta\rangle}$, while $[\![B]\!]_{\eta\langle X \leftarrow [\![C]\!]\eta\rangle} = [\![B\{X \leftarrow C\}]\!]_\eta$ by a standard substitution lemma. Moreover, $[\![b]\!]_\rho = [\![b(C)]\!]_\rho$, $[\![b']\!]_{\rho'} = [\![b'(C)]\!]_{\rho'}$. The hypothesis yields $([\![b]\!]_\rho, [\![b']\!]_{\rho'}) \in [\![\forall(X<:A)B]\!]_\eta$, and hence

$$([\![b(C)]\!]_\rho, [\![b'(C)]\!]_{\rho'}) \in [\![B\{X \leftarrow C\}]\!]_\eta$$

5. For (Eq Pack<:):

$$\frac{E \vdash C <: A' \qquad E \vdash A' <: A \qquad E, X<:A' \vdash B'\{X\} <: B\{X\} \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash (\text{pack } X<:A=C, b\{X\}:B\{X\}) \ \leftrightarrow \ (\text{pack } X<:A'=C, b'\{X\}:B'\{X\}) : \exists(X<:A)B\{X\}}$$

This case is similar to that for (Eq Appl2<:), noting here that $[\![B']\!]_{\eta\langle X \leftarrow [\![C]\!]\eta\rangle} \subseteq [\![\exists(X<:A)B]\!]_\eta$ and that $[\![b\{C\}]\!]_\rho = [\![\text{pack } X<:A=C, b\{X\}:B\{X\}]\!]_\rho$ and $[\![b'\{C\}]\!]_{\rho'} = [\![\text{pack } X<:A'=C, b'\{X\}:B'\{X\}]\!]_{\rho'}$.

6. For (Eq Open<:):

$$\frac{E \vdash c \leftrightarrow c' : \exists(X<:A)B \qquad E \vdash D \qquad E, X<:A, x:B \vdash d \leftrightarrow d' : D}{E \vdash (\text{open } c \text{ as } X<:A, x:B \text{ in } d:D) \ \leftrightarrow \ (\text{open } c' \text{ as } X<:A, x:B \text{ in } d':D) : D}$$

By the first hypothesis, we have that $([\![c]\!]_\rho, [\![c']\!]_{\rho'}) \in \sqcup_{R \in \textbf{CUPER}, R \subseteq [\![A]\!]_\eta} [\![B]\!]_{\eta\langle X \leftarrow R\rangle}$, and we wish to derive that $([\![d\{x \leftarrow c\}]\!]_\rho, [\![d'\{x \leftarrow c'\}]\!]_{\rho'}) \in [\![D]\!]_\eta$, that is, by a standard substitution lemma, that

$(\llbracket d\rrbracket_{\rho\langle x\leftarrow\llbracket c\rrbracket\rho\rangle}, \llbracket d'\rrbracket_{\rho'\langle x\leftarrow\llbracket c'\rrbracket\rho'\rangle}) \in \llbracket D\rrbracket_\eta$. By Lemma 9.3.3-1, it suffices to prove that if $(v,v') \in \llbracket B\rrbracket_{\eta\langle X\leftarrow R\rangle}$ for some $R \subseteq \llbracket A\rrbracket_\eta$ then $(\llbracket d\rrbracket_{\rho\langle x\leftarrow v\rangle}, \llbracket d'\rrbracket_{\rho'\langle x\leftarrow v'\rangle}) \in \llbracket D\rrbracket_\eta$. If η and (ρ,ρ') are consistent with E, then $\eta\langle X\leftarrow R\rangle$ and $(\rho\langle x\leftarrow v\rangle, \rho'\langle x\leftarrow v'\rangle)$ are consistent with E,X<:A,x:B. The third hypothesis yields $(\llbracket d\rrbracket_{\rho\langle x\leftarrow v\rangle}, \llbracket d'\rrbracket_{\rho'\langle x\leftarrow v'\rangle}) \in \llbracket D\rrbracket_{\eta\langle X\leftarrow R\rangle}$, and since X cannot occur in D (because $E \vdash D$) we obtain $(\llbracket d\rrbracket_{\rho\langle x\leftarrow v\rangle}, \llbracket d'\rrbracket_{\rho'\langle x\leftarrow v'\rangle}) \in \llbracket D\rrbracket_\eta$.

7. For (Eval Beta):

$$\frac{E \vdash \lambda(x:A)b : A\to B \qquad E \vdash a : A}{E \vdash (\lambda(x:A)b)(a) \;\leftrightarrow\; b\{x\leftarrow a\} : B}$$

Consider η and (ρ,ρ') consistent with E. As in the case of application, the hypotheses yield $(\llbracket(\lambda(x:A)b)(a)\rrbracket_\rho, \llbracket(\lambda(x:A)b)(a)\rrbracket_{\rho'}) \in \llbracket B\rrbracket_\eta$. Because of the form of $\lambda(x:A)b$, we have that $\llbracket\lambda(x:A)b\rrbracket_{\rho'} \in (D\to D)$ and hence $\llbracket(\lambda(x:A)b)(a)\rrbracket_{\rho'} = \llbracket b\rrbracket_{\rho'\langle x\leftarrow\llbracket a\rrbracket\rho'\rangle}$. A standard substitution lemma yields $\llbracket(\lambda(x:A)b)(a)\rrbracket_{\rho'} = \llbracket b\{x\leftarrow a\}\rrbracket_{\rho'}$. Therefore, $(\llbracket(\lambda(x:A)b)(a)\rrbracket_\rho, \llbracket b\{x\leftarrow a\}\rrbracket_{\rho'}) \in \llbracket B\rrbracket_\eta$.

8. For (Eval Eta):

$$\frac{E \vdash b : A\to B \qquad x\notin dom(E)}{E \vdash \lambda(x:A)b(x) \;\leftrightarrow\; b : A\to B}$$

Let $(v,v') \in \llbracket A\rrbracket_\eta$. The hypothesis yields that $\llbracket b\rrbracket_\rho, \llbracket b\rrbracket_{\rho'} \in (D\to D)$. We obtain that $\llbracket\lambda(x:A)b(x)\rrbracket_\rho = \lambda(u)\llbracket b(x)\rrbracket_{\rho\langle x\leftarrow u\rangle}$ and hence $\llbracket\lambda(x:A)b(x)\rrbracket_\rho = \lambda(u)\llbracket b\rrbracket_{\rho\langle x\leftarrow u\rangle}(\llbracket x\rrbracket_{\rho\langle x\leftarrow u\rangle})$ and, since x is assumed not to occur free in b, $\llbracket\lambda(x:A)b(x)\rrbracket_\rho = \lambda(u)\llbracket b\rrbracket_\rho(u)$. Furthermore, the hypothesis gives $(\llbracket b\rrbracket_\rho(v), \llbracket b'\rrbracket_{\rho'}(v')) \in \llbracket B\rrbracket_\eta$, and hence $(\llbracket\lambda(x:A)b(x)\rrbracket_\rho(v), \llbracket b'\rrbracket_{\rho'}(v')) \in \llbracket B\rrbracket_\eta$. Finally, we have

$$(\llbracket\lambda(x:A)b(x)\rrbracket_\rho, \llbracket b'\rrbracket_{\rho'}) \in \llbracket A\rrbracket_\eta\to\llbracket B\rrbracket_\eta$$

that is,

$$(\llbracket\lambda(x:A)b(x)\rrbracket_\rho, \llbracket b'\rrbracket_{\rho'}) \in \llbracket A\to B\rrbracket_\eta$$

9. For (Eval Beta2<:):

$$\frac{E \vdash \lambda(X<:A)b : \forall(X<:A)B \qquad E \vdash C <: A}{E \vdash (\lambda(X<:A)b)(C) \;\leftrightarrow\; b\{X\leftarrow C\} : B\{X\leftarrow C\}}$$

As in the case of second-order application, we use that $\llbracket\forall(X<:A)B\rrbracket_\eta \subseteq \llbracket B\{X\leftarrow C\}\rrbracket_\eta$. Moreover, $\llbracket(\lambda(X<:A)b)(C)\rrbracket_\rho = \llbracket\lambda(X<:A)b\rrbracket_\rho$ and $\llbracket b\{X\leftarrow C\}\rrbracket_{\rho'} = \llbracket\lambda(X<:A)b\rrbracket_{\rho'}$. Hence, using the hypothesis, we obtain $(\llbracket(\lambda(X<:A)b)(C)\rrbracket_\rho, \llbracket b\{X\leftarrow C\}\rrbracket_{\rho'}) \in \llbracket B\{X\leftarrow C\}\rrbracket_\eta$.

10. For (Eval Eta2<:):

$$\frac{E \vdash b : \forall(X<:A)B \qquad X\notin dom(E)}{E \vdash \lambda(X<:A)b(X) \;\leftrightarrow\; b : \forall(X<:A)B}$$

This case is trivial, since $\llbracket\lambda(X<:A)b(X)\rrbracket_\rho = \llbracket b\rrbracket_\rho$.

11. For (Eval Unpack<:):

$$\frac{E \vdash b\{C\} : B\{C\} \qquad E \vdash C <: A \qquad E \vdash D \qquad E,X<:A,x:B\{X\} \vdash d\{X,x\} : D}{E \vdash (\text{open (pack } X<:A=C,\, b\{X\}:B\{X\}) \text{ as } X<:A,x:B\{X\} \text{ in } d\{X,x\}:D) \;\leftrightarrow\; d\{C,b\{C\}\} : D}$$

We have that $[\![$open (pack X<:A=C, b{X}:B{X}) as X<:A,x:B{X} in d{X,x}:D$]\!]_\rho$ is $[\![d\{x\leftarrow b\}]\!]_\rho$. By a standard substitution lemma, it is then sufficient to prove that $([\![d]\!]_{\rho\langle x\leftarrow[\![b]\!]_\rho\rangle}, [\![d']\!]_{\rho'\langle x\leftarrow[\![b']\!]_{\rho'}\rangle}) \in [\![D]\!]_\eta$. If η and (ρ,ρ') are consistent with E, then $\eta\langle X\leftarrow[\![C]\!]_\eta\rangle$ and $(\rho\langle x\leftarrow[\![b]\!]_\rho\rangle, \rho'\langle x\leftarrow[\![b]\!]_{\rho'}\rangle)$ are consistent with E,X<:A,x:B, since $([\![b]\!]_\rho, [\![b]\!]_{\rho'}) \in [\![B\{C\}]\!]_\eta$ by the first hypothesis and $[\![C]\!]_\eta \subseteq [\![A]\!]_\eta$ by the second one. Therefore, the last hypothesis yields that $([\![d]\!]_{\rho\langle x\leftarrow[\![b]\!]_\rho\rangle}, [\![d']\!]_{\rho'\langle x\leftarrow[\![b']\!]_{\rho'}\rangle}) \in [\![D]\!]_{\eta\langle X\leftarrow[\![C]\!]_\eta\rangle}$. But since X cannot occur in D (because E $\vdash$ D) we obtain $([\![d]\!]_{\rho\langle x\leftarrow[\![b]\!]_\rho\rangle}, [\![d']\!]_{\rho'\langle x\leftarrow[\![b']\!]_{\rho'}\rangle}) \in [\![D]\!]_\eta$.

12. For (Eval Repack<:):

$$\frac{E \vdash b : \exists(X<:A)B\{X\} \qquad E,y:\exists(X<:A)B\{X\} \vdash d\{y\} : D}{E \vdash (\text{open } b \text{ as } X<:A,x:B\{X\} \text{ in } d\{\text{pack } X'<:A=X,\ x:B\{X'\}\}:D) \ \leftrightarrow\ d\{b\} : D}$$

We have that $[\![$open b as X<:A,x:B{X} in d{pack X'<:A=X, x:B{X'}}:$\exists$(X'<:A)B{X'}$]\!]_\rho$ is $[\![d\{b\}]\!]_\rho$. The result thus follows from the hypotheses, using a standard substitution lemma.

13. For (Eq Top):

$$\frac{E \vdash a : A \qquad E \vdash b : B}{E \vdash a \leftrightarrow b : \text{Top}}$$

From the hypotheses we obtain that $[\![a]\!]_\rho$ and $[\![b]\!]_\rho$. are not *. Hence $([\![a]\!]_\rho, [\![b]\!]_{\rho'}) \in [\![\text{Top}]\!]_\eta$.

14. For (Eq Object): (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i\in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \ \leftrightarrow\ [l_i=\varsigma(x_i:A)b_i'\ ^{i\in 1..n}] : A}$$

The first hypothesis yields

$$([\![\lambda(x_i:A)b_i]\!]_\rho, [\![\lambda(x_i:A)b_i']\!]_{\rho'}) \in [\![A\rightarrow B_i]\!]_\eta$$

(The argument is a special case of the one in (1).) According to the definition of the semantics, this means that

$$([\![\lambda(x_i:A)b_i]\!]_\rho, [\![\lambda(x_i:A)b_i']\!]_{\rho'}) \in [[m_i : [\![B_i]\!]_\eta\ ^{i\in 1..n}]]\rightarrow[\![B_i]\!]_\eta$$

for all $i \in 1..n$. Proposition 9.3.5-5 yields

$$(\langle\langle l_i=[\![\lambda(x_i:A)b_i]\!]_\rho\ ^{i\in 1..n}\rangle\rangle, \langle\langle l_i=[\![\lambda(x_i:A)b_i']\!]_{\rho'}\ ^{i\in 1..n}\rangle\rangle) \in [[l_i : [\![B_i]\!]_\eta\ ^{i\in 1..n}]]$$

that is,

$$([\![[l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}]]\!]_\rho, [\![[l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}]]\!]_{\rho'}) \in [\![[l_i : B_i\ ^{i\in 1..n}]]\!]_\eta$$

15. For (Eq Sub Object): (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $A' \equiv [l_i:B_i\ ^{i\in 1..n}, l_j:B_j\ ^{j\in n+1..m}]$)

$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i\in 1..n \qquad E, x_j:A' \vdash b_j : B_j \quad \forall j\in n+1..m}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \ \leftrightarrow\ [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}] : A}$$

The first hypothesis yields

$$([\![\lambda(x_i:A)b_i]\!]_\rho, [\![\lambda(x_i:A)b_i]\!]_{\rho'}) \in [\![A\rightarrow B_i]\!]_\eta$$

(The argument is a special case of the one in (1).) According to the definition of the semantics, this means that

$$(\llbracket\lambda(x_i{:}A)b_i\rrbracket_\rho, \llbracket\lambda(x_i{:}A)b_i\rrbracket_{\rho'}) \in [[m_i : \llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]\to\llbracket B_i\rrbracket_\eta$$

for all $i \in 1..n$. Proposition 9.3.5-5 yields

$$(\langle\langle l_i=\llbracket\lambda(x_i{:}A)b_i\rrbracket_\rho \ ^{i\in 1..n}\rangle\rangle, \langle\langle l_i=\llbracket\lambda(x_i{:}A)b_i\rrbracket_{\rho'} \ ^{i\in 1..n}, l_j=\llbracket\lambda(x_j{:}A')b_j\rrbracket_{\rho'} \ ^{j\in n+1..m}\rangle\rangle)$$
$$\in [[l_i : \llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]$$

that is,

$$(\llbracket[l_i=\varsigma(x_i{:}A)b_i \ ^{i\in 1..n}]\rrbracket_\rho, \llbracket[l_i=\varsigma(x_i{:}A')b_i \ ^{i\in 1..n}, l_j=\varsigma(x_j{:}A')b_j \ ^{j\in n+1..m}]\rrbracket_{\rho'}) \in \llbracket[l_i : B_i \ ^{i\in 1..n}]\rrbracket_\eta$$

Note that the second hypothesis is of no use to us in this proof.

16. For (Eq Select):

$$\frac{E \vdash a \leftrightarrow a' : [l_i{:}B_i \ ^{i\in 1..n}] \qquad j\in 1..n}{E \vdash a.l_j \ \leftrightarrow \ a'.l_j : B_j}$$

The hypothesis yields $(\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'}) \in \llbracket[l_i{:}B_i \ ^{i\in 1..n}]\rrbracket_\eta$, so $(\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'}) \in [[l_i{:}\llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]$. By Proposition 9.3.5-6, $\llbracket a\rrbracket_\rho = \llbracket a'\rrbracket_{\rho'} = \perp$ or $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$. If $\llbracket a\rrbracket_\rho = \llbracket a'\rrbracket_{\rho'} = \perp$ then immediately $\llbracket a.l_j\rrbracket_\rho = \llbracket a'.l_j\rrbracket_{\rho'} = \perp$ so $(\llbracket a.l_j\rrbracket_\rho, \llbracket a'.l_j\rrbracket_{\rho'}) \in \llbracket B_j\rrbracket_\eta$. Otherwise, $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$. By Proposition 9.3.5-7, $\llbracket a\rrbracket_\rho(l_j), \llbracket a'\rrbracket_{\rho'}(l_j) \in (D\to D)$. By Proposition 9.3.5-8,

$$(\llbracket a\rrbracket_\rho(l_j)(\llbracket a\rrbracket_\rho), \llbracket a'\rrbracket_{\rho'}(l_j)(\llbracket a'\rrbracket_{\rho'})) \in \llbracket B_j\rrbracket_\eta$$

Since $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$ and $\llbracket a\rrbracket_\rho(l_j), \llbracket a'\rrbracket_{\rho'}(l_j) \in (D\to D)$, this last result means

$$(\llbracket a.l_j\rrbracket_\rho, \llbracket a'.l_j\rrbracket_{\rho'}) \in \llbracket B_j\rrbracket_\eta$$

17. For (Eq Override): (where $A \equiv [l_i{:}B_i \ ^{i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \qquad E, x{:}A \vdash b \leftrightarrow b' : B_j \qquad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b \ \leftrightarrow \ a'.l_j \Leftarrow \varsigma(x{:}A)b' : \ A}$$

One hypothesis yields $(\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'}) \in \llbracket[l_i{:}B_i \ ^{i\in 1..n}]\rrbracket_\eta$, that is, $(\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'}) \in [[l_i{:}\llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]$. The second one yields $(\llbracket\lambda(x{:}A)b\rrbracket_\rho, \llbracket\lambda(x{:}A)b'\rrbracket_{\rho'}) \in \llbracket[l_i{:}B_i \ ^{i\in 1..n}]\to C_j\rrbracket_\eta$, that is,

$$(\llbracket\lambda(x{:}A)b\rrbracket_\rho, \llbracket\lambda(x{:}A)b'\rrbracket_{\rho'}) \in [[l_i : \llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]\to\llbracket B_j\rrbracket_\eta$$

(The argument is a special case of the one in (1).) By Proposition 9.3.5-6, $\llbracket a\rrbracket_\rho = \llbracket a'\rrbracket_{\rho'} = \perp$ or $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$. If $\llbracket a\rrbracket_\rho = \llbracket a'\rrbracket_{\rho'} = \perp$, then immediately

$$\llbracket a.l_j \Leftarrow \varsigma(x{:}A)\ b\rrbracket_\rho = \llbracket a'.l_j \Leftarrow \varsigma(x{:}A)\ b'\rrbracket_{\rho'} = \perp$$

so $(\llbracket a.l_j \Leftarrow \varsigma(x{:}A)\ b\rrbracket_\rho, \llbracket a'.l_j \Leftarrow \varsigma(x{:}A)\ b'\rrbracket_{\rho'}) \in \llbracket[l_i : B_i \ ^{i\in 1..n}]\rrbracket_\eta$. Otherwise, $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$. Proposition 9.3.5-9 yields

$$(\llbracket a\rrbracket_\rho\{l_j\leftarrow\llbracket\lambda(x{:}A)b\rrbracket_\rho\}, \llbracket a'\rrbracket_\rho\{l_j'\leftarrow\llbracket\lambda(x{:}A)b'\rrbracket_{\rho'}\}) \in [[l_i : \llbracket B_i\rrbracket_\eta \ ^{i\in 1..n}]]$$

Since $\llbracket a\rrbracket_\rho, \llbracket a'\rrbracket_{\rho'} \in (L\to D)$, this last result means

$$(\llbracket a.l_j \Leftarrow \varsigma(x{:}A)\ b\rrbracket_\rho, \llbracket a'.l_j \Leftarrow \varsigma(x{:}A)\ b'\rrbracket_{\rho'}) \in \llbracket[l_i : B_i \ ^{i\in 1..n}]\rrbracket_\eta$$

18. For (Eval Select): (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \ \leftrightarrow\ b_j\{x_j \leftarrow a\} : B_j}$$

The hypothesis yields $([\![a]\!]_\rho, [\![a]\!]_{\rho'}) \in [\![A]\!]_\eta$, that is, $(\langle\langle l_i = [\![\lambda(x_i:A')b_i]\!]_\rho\rangle\rangle, \langle\langle l_i = [\![\lambda(x_i:A')b_i]\!]_{\rho'}\rangle\rangle) \in [[l_i : [\![B_i]\!]_\eta]]$. The argument of (16) gives

$$([\![a.l_j]\!]_\rho, [\![a.l_j]\!]_{\rho'}) \in [\![B_j]\!]_\eta$$

In addition, $[\![a]\!]_{\rho'} \in (L \to D)$ and $[\![\lambda(x_j:A')b_j]\!]_{\rho'} \in (D \to D)$. Hence,

$$[\![a.l_j]\!]_{\rho'} = [\![\lambda(x_j:A')b_j]\!]_{\rho'}([\![a]\!]_{\rho'})$$

so

$$[\![a.l_j]\!]_{\rho'} = [\![b_j]\!]_{\rho'\langle x_j \leftarrow [\![a]\!]_{\rho'}\rangle}$$

A standard substitution lemma yields

$$[\![a.l_j]\!]_{\rho'} = [\![b_j\{x_j \leftarrow a\}]\!]_{\rho'}$$

and hence

$$([\![a.l_j]\!]_\rho, [\![b_j\{x_j \leftarrow a\}]\!]_{\rho'}) \in [\![B_j]\!]_\eta$$

19. For (Eval Override): (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \ \leftrightarrow\ [l_j = \varsigma(x:A')b,\ l_i = \varsigma(x_i:A')b_i\ ^{i\in(1..n+m)-\{j\}}] :\ A}$$

The first hypothesis yields

$$([\![a]\!]_\rho, [\![a]\!]_{\rho'}) \in [\![A]\!]_\eta$$

that is, $(\langle\langle l_i = [\![\lambda(x_i:A)b_i]\!]_\rho\rangle\rangle, \langle\langle l_i = [\![\lambda(x_i:A)b_i]\!]_{\rho'}\rangle\rangle) \in [[l_i : [\![B_i]\!]_\eta]]$. The argument of (17) gives

$$([\![a.l_j \Leftarrow \varsigma(x:A)\ b]\!]_\rho, [\![a.l_j \Leftarrow \varsigma(x:A)\ b]\!]_{\rho'}) \in [\![A]\!]_\eta$$

In addition, $[\![a]\!]_{\rho'} \in (L \to D)$. Hence,

$$[\![a.l_j \Leftarrow \varsigma(x:A)\ b]\!]_{\rho'} = [\![[l_{i,i\neq j} = \varsigma(x_i:A')b_i,\ m_j = \varsigma(x:A')b]]\!]_{\rho'}$$

and

$$([\![a.l_j \Leftarrow \varsigma(x:A)\ b]\!]_\rho, [\![[l_{i,i\neq j} = \varsigma(x_i:A')b_i,\ m_j = \varsigma(x:A')b]]\!]_{\rho'}) \in [\![A]\!]_\eta$$

20. For (Eq Subsumption):

$$\frac{E \vdash a \leftrightarrow a' : A \qquad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

This rule is immediately sound, since subtyping is simply inclusion.

21. For (Sub Object):

$$\frac{E \vdash B_i \qquad \forall i \in 1..n+m}{E \vdash [l_i{:}B_{i\ i\in 1..n+m}] <: [l_i{:}B_{i\ i\in 1..n}]}$$

This follows immediately from the definition of the semantics and Proposition 9.3.5-4.

From these verifications we derive the following soundness results:

Theorem 9.4.3-1

Assume that η and (ρ, ρ') are consistent with E. Then, for derivations in $\mathbf{FOb}_{<:\mu}$:

If	$E \vdash A$	then	$[\![A]\!]_\eta \in \mathbf{CUPER}$.
If	$E \vdash A <: B$	then	$[\![A]\!]_\eta \subseteq [\![B]\!]_\eta$.
If	$E \vdash a : A$	then	$([\![a]\!]_\rho, [\![a]\!]_{\rho'}) \in [\![A]\!]_\eta$.
If	$E \vdash a \leftrightarrow a' : A$	then	$([\![a]\!]_\rho, [\![a']\!]_{\rho'})$ in $[\![A]\!]_\eta$.

□

By reversing the last two implications, it follows that:

- If $[\![a]\!]_\rho = *$ then $E \nvdash a{:}A$, therefore the type theory is sound.
- If $[\![a]\!]_\rho \neq [\![a']\!]_{\rho'}$ then $E \nvdash a \leftrightarrow a'{:}A$, therefore the equational theory is consistent.

10. A Translation and its Coherence

The interpretation of the previous section suggests many translations of the object calculus into other calculi. Finding a good translation proves rather delicate, however: we should avoid introducing new subtyping relations or new subtyping properties in our target calculus, lest it becomes almost as special as the source calculus. We would like a syntactic explanation of objects in terms of more standard constructs, with regular rules and general semantics.

Breazu-Tannen *et al.* [Breazu-Tannen, *et al.* 1991] have explored an interpretation of subtyping in terms of implicit coercions. Their source calculus is an extension of $\boldsymbol{F}_{<:}$ with recursion and records. Their target calculus is an extension of System $\boldsymbol{F}$ with recursion and records. We adopt their approach, adding existential quantifiers and rows to their target calculus.

Before describing our target calculus, we briefly review the method of Breazu-Tannen *et al.* In this method, each type A of the source calculus is translated into a type A* of the target calculus. For example, $(\forall(X<:A)B)^*$ is $\forall(X)(X \to A^*) \to B^*$, the type of polymorphic functions that produce a B* from a coercion from X into A*. Note how the bounded quantifier is replaced by a standard quantifier and some arrows. Correspondingly, a subtype judgment $A <: B$ is translated into a coercion function of type $A^* \to B^*$, by induction on the derivation of A<:B. For example, A<:A yields the identity function $\lambda(x:A^*)x$. Finally, a term b of type B is translated into a term b^B of type B*, by induction on the derivation of b:B. For example, if b:B is derived from b:A and A<:B by subsumption, then b^B is $f(b^A)$, where f is the coercion for A<:B. A typing b:B can be obtained via several different derivations; a coherence theorem guarantees that all the resulting translations are equivalent. The details of the method are involved; we will describe them further only in the context of our calculus.

10.1 The Target Calculus

We define the calculus that is the target of our translation. It is an extension of System **F** with recursion, records, and rows. Thus, all of its constructs are standard except for rows, which we explain next.

A row is a set of labels and types $l_i{:}B_i\ ^{i\in 1..n}$, much like an incomplete record type. A row R has kind $\uparrow(l_i\ ^{i\in 1..n})$ if it is missing the labels $l_i\ ^{i\in 1..n}$. If a row R has kind $\uparrow()$, then $\langle R\rangle$ is a record type. The rows we use are simpler than the ones treated in [Cardelli 1991], since we do not need rows at the value level. Rows are a convenient but not an essential addition: they can be encoded in $\boldsymbol{F}_{<:}$ [Cardelli 1991], and hence in $\boldsymbol{F}$. For simplicity we do not consider this reduction further. We note only that the reduction yields that the target calculus is sound.

The full target calculus appears in appendix F. Here we explain some of the most interesting constructs and rules.

The kinds K of the calculus are Type (the kind of all types) and the kinds of rows $\uparrow(l_i\ ^{i\in 1..n})$. The types have the forms X, $\langle R\rangle$, $A\to B$, $\forall(X::K)B$, $\exists(X::K)B$, and $\mu(X)A$. We write $E\vdash A$ as an abbreviation for $E\vdash A::\text{Type}$, which means that A is a type; $E\vdash R::\uparrow(l_1..l_n)$ means that R is a row missing $l_1..l_n$.

(Type Record) $\dfrac{E\vdash R::\uparrow()}{E\vdash\langle R\rangle}$ (Row $\emptyset$) $\dfrac{E\vdash\diamond}{E\vdash\emptyset::\uparrow(l_1..l_n)}$ (Row Cons) $\dfrac{E\vdash R::\uparrow(l,l_1..l_n)\quad E\vdash B}{E\vdash l{:}B,R::\uparrow(l_1..l_n)}$

(Val Record) $\dfrac{E\vdash b_i:B_i\quad \forall i\in 1..n}{E\vdash\langle l_i=b_i\ ^{i\in 1..n}\rangle:\langle l_i{:}B_i\ ^{i\in 1..n},\emptyset\rangle}$ (Val Select) $\dfrac{E\vdash a:\langle l{:}B,R\rangle}{E\vdash a\cdot l:B}$ (Val Update) $\dfrac{E\vdash a:\langle l{:}B,R\rangle\quad E\vdash b:B}{E\vdash a\cdot l:=b:\langle l{:}B,R\rangle}$

A record type $\langle R\rangle$ can be formed from a row R that is missing no labels. Such a row can be assembled from the empty row $\emptyset$, which lacks any set of labels, by successive additions of components l:B. A record of type $\langle l_i{:}B_i\ ^{i\in 1..n},\emptyset\rangle$ can be built from components $b_i{:}B_i$. The rules for record selection and update make use of rows, and are otherwise straightforward. Note that we have no way of extending existing records.

10.2 The Translation of $\mathbf{Ob}_{1<:}$

Next we give the translation from $\mathbf{Ob}_{1<:}$ into the target calculus. We omit the treatment of constants. Recursion and quantification are discussed later.

A derivation Ψ of a judgment $\Im$ is written $\Psi_1,...,\Psi_n \rhd_R \Im$, where $n\geq 0$, R is the last rule of Ψ, and $\Psi_1,...,\Psi_n$ are the derivations of the assumptions of R. Further, $\overline{\Psi}\rhd_R\Im$ stands for a derivation of $\Im$ via R, and $\overline{\Psi}\rhd\Im$ for an arbitrary derivation of $\Im$.

We define the following translations for the types and judgments of $\mathbf{Ob}_{1<:}$:

A^*	is a type in the target calculus of appendix F
$(\overline{\Psi}\rhd E\vdash A<:B)^*$	is a term of type $A^*\to B^*$ in the target calculus
$(\overline{\Psi}\rhd E\vdash a:A)^*$	is a term of type A^* in the target calculus

For object types, an existential quantifier over rows replaces the union that appears in the semantics; it is convenient to rearrange a record of functions with equal domain into a function that returns a record:

$$\text{let } \{\!|l_i{:}B_i\ ^{i\in 1..n},X|\!\} = \mu(Y)\ Y\to\langle l_i{:}B_i\ ^{i\in 1..n},X\rangle \qquad Y\notin FV(B_j),\ Y\neq X$$

$$[l_i{:}B_i\ ^{i\in 1..n}]^* = \exists(X::\uparrow(l_i\ ^{i\in 1..n}))\ \{\!|l_i{:}B_i{}^*\ ^{i\in 1..n},X|\!\} \qquad X\notin FV(B_j)$$

The largest type, Top, is mapped to the empty record type; any terminal type would do.

$\mathrm{Top}^* \triangleq \langle\rangle$ (the empty record type)

The translation of subtypings and terms is by induction on derivations. For subtypings, we set:

$$(\overline{\Psi} \rhd_{(\text{Sub Refl})} E \vdash A <: A)^* \triangleq \lambda(x{:}A^*)x$$

$$((\overline{\Psi}_1 \rhd E \vdash A <: B,\ \overline{\Psi}_2 \rhd E \vdash B <: C) \rhd_{(\text{Sub Trans})} E \vdash A <: C)^* \triangleq$$
$$\lambda(x{:}A^*)\,(\overline{\Psi}_2 \rhd E \vdash B <: C)^*\,((\overline{\Psi}_1 \rhd E \vdash A <: B)^*\,(x))$$

$$(\overline{\Psi} \rhd_{(\text{Sub Top})} E \vdash A <: \mathrm{Top})^* \triangleq \lambda(x{:}A^*)\langle\rangle$$

$$(\overline{\Psi} \rhd_{(\text{Sub Object})} E \vdash [l_i{:}B_i\ ^{i\in 1..n+m}] <: [l_i{:}B_i\ ^{i\in 1..n}])^* \triangleq$$
$$\lambda(o{:}[l_i{:}B_i\ ^{i\in 1..n+m}]^*)$$
$$\quad \text{open } o \text{ as } X{::}\uparrow(l_i\ ^{i\in 1..n+m}),\ x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n+m}, X|\!\}$$
$$\quad \text{in pack } X'{::}\uparrow(l_i\ ^{i\in 1..n})=(l_i{:}B_i^*\ ^{i\in n+1..m}, X),\ x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X'|\!\}$$

An application of reflexivity yields the identity coercion. Transitivity corresponds to composition of coercions. Inclusion in Top yields the canonical coercion into the empty record type. Inclusion of object types produces a coercion that repackages type information while leaving the underlying records unchanged. Note that if the empty record type is interpreted as containing every value, then the coercions into it can be implemented as identity functions. Then all the coercions generated by the translation are implemented as identity functions as well.

Similarly, each derivation of a value judgment $E \vdash a : A$ is translated into a term of type A^*.

$$((\overline{\Psi}_1 \rhd E \vdash a : A,\ \overline{\Psi}_2 \rhd E \vdash A <: B) \rhd_{(\text{Val Subsumpiton})} E \vdash a : B)^* \triangleq$$
$$(\overline{\Psi}_2 \rhd E \vdash A <: B)^*\,(\overline{\Psi}_1 \rhd E \vdash a : A)^*$$

$$(\overline{\Psi} \rhd_{(\text{Val x})} E', x{:}A, E'' \vdash x{:}A)^* \triangleq x$$

$$((\overline{\Psi}_i \rhd E, x_i{:}A \vdash b_i : B_i) \rhd_{(\text{Val Object})} E \vdash [l_i = \varsigma(x_i{:}A)b_i\ ^{i\in 1..n}] : A)^* \triangleq$$
$$\text{pack } X{::}\uparrow(l_i\ ^{i\in 1..n})=\emptyset,$$
$$\quad \mathrm{fold}(\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\},$$
$$\quad\quad \lambda(x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\})$$
$$\quad\quad\quad \langle l_i = (\overline{\Psi}_i \rhd E, x_i{:}A \vdash b_i : B_i)^*$$
$$\quad\quad\quad\quad \{x_i \leftarrow \text{pack } X{::}\uparrow(l_i\ ^{i\in 1..n})=\emptyset,\ x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\}\}\ ^{i\in 1..n}\rangle)$$
$$\quad : \{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\}$$

$$((\overline{\Psi} \rhd E \vdash a : [l_i{:}B_i\ ^{i\in 1..n}]) \rhd_{(\text{Val Select})} E \vdash a.l_j : B_j)^* \triangleq$$
$$\text{open } (\overline{\Psi} \rhd E \vdash a : [l_i{:}B_i\ ^{i\in 1..n}])^* \text{ as } X{::}\uparrow(l_i\ ^{i\in 1..n}),\ x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\}$$
$$\text{in } \mathrm{unfold}(x)(x)\cdot l_j$$

$$((\overline{\Psi}_1 \rhd E \vdash a : A,\ \overline{\Psi}_2 \rhd E, x{:}A \vdash b : B_j) \rhd_{(\text{Val Override})} E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A)^* \triangleq$$
$$\text{open } (\overline{\Psi}_1 \rhd E \vdash a : [l_i{:}B_i\ ^{i\in 1..n}])^* \text{ as } X{::}\uparrow(l_i\ ^{i\in 1..n}),\ x{:}\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X|\!\}$$
$$\text{in pack } X'{::}\uparrow(l_i\ ^{i\in 1..n})=X,$$
$$\quad \mathrm{fold}(\{\!|l_i{:}B_i^*\ ^{i\in 1..n}, X'|\!\},$$

$$\lambda(x':\{\!|l_i:B_i^{\,i\in 1..n}, X'|\!\})$$
$$\quad \mathrm{unfold}(x)(x')\cdot l_j :=$$
$$\quad\quad (\overline{\Psi}_2 \triangleright E, x:A \vdash b : B_j)^*$$
$$\quad\quad\quad \{x \leftarrow \mathrm{pack}\ X'' :: \uparrow(l_i^{\,i\in 1..n}) = X',\ x':\{\!|l_i:B_i^{\,i\in 1..n}, X''|\!\}\})$$
$$: \{\!|l_i:B_i^{\,i\in 1..n}, X'|\!\}$$

Subsumption is treated as the application of a coercion. The rule for variables is trivial. An object becomes, in essence, a function that returns a record. An object selection becomes a self-application followed by a record selection. Object override is reduced to record update. The translation of each object construct requires a careful use of pack, open, fold, and unfold, guided by the type structure.

Further, the congruence rules and evaluation rules for objects are validated through the translation. So are the equational rules related to subtyping (Eq Subsumption) and (Eq Top), but not (Eq Sub Object). The soundness proof is by heavy calculations, using a straightforward substitution lemma. Proving the soundness of (Eq Sub Object) in the target calculus may require additional principles, such as a bisimulation rule [Aczel, Mendler 1989].

10.3 Coherence for $\mathbf{Ob}_{1<:}$

As the translation is defined by induction on derivations, we face a coherence problem analogous to that of Breazu-Tannen *et al.*. We obtain two coherence results for $\mathbf{Ob}_{1<:}$ (omitting constants). The results imply that the terms obtained by mapping two different derivations of the same judgment are in fact provably equal in the target calculus. Therefore, the translation provides an unambiguous semantics of judgments.

Theorem 10.3-1 (Coherence of coercions)

If f and g are the translations of two $\mathbf{Ob}_{1<:}$ derivations that end with the same subtype judgment, then f and g are provably equal in the target calculus.

Proof

We exploit the fact that it is easy to know which subtypings are provable in this simple system. We prove that if the judgment is $E \vdash [l_i:B_i^{\,i\in 1..n+m}] <: [l_i:B_i^{\,i\in 1..n}]$ then any coercion f equals the coercion given for (Sub Object), and if the judgment is $E \vdash A <: \mathrm{Top}$ then any coercion f equals the coercion given for (Sub Top) $\lambda(x:A^*)\langle\rangle$. The proof is by induction on subtyping derivations:

(Sub Refl) $E \vdash A <: A$.

If A is $[l_i:B_i^{\,i\in 1..n}]$, then we need to check that the coercion given for (Sub Object) equals $\lambda(x:A^*)x$. This follows from the η rule for existentials (Eval Repack).

If A is Top, then we need to check that $\lambda(x:A^*)\langle\rangle$ equals $\lambda(x:A^*)x$. This holds by the η rule for records (Eval Rerecord): Top* is $\langle\rangle$, and any x in $\langle\rangle$ equals the empty record $\langle\rangle$.

(Sub Trans) $E \vdash A <: C$ via $E \vdash A <: B$ and $E \vdash B <: C$.

If A is $[l_i:B_i^{\,i\in 1..n+m+p}]$, B is $[l_i:B_i^{\,i\in 1..n+m}]$, and C is $[l_i:B_i^{\,i\in 1..n}]$, then we need to check that the direct coercion from A* to C* is equal to the composition of the coercions from A* to B* and from B* to C*. All these coercions are given by the translation of (Sub Object). Here $A^* = \exists(X::\uparrow(l_i^{\,i\in 1..m+n+p}))\{\!|l_i:B_i^{\,i\in 1..n+m+p}, X|\!\}$. By the η rule for existentials, we need to test the equality only for a term of the form $\mathrm{pack}\ X::\uparrow(l_i^{\,i\in 1..n+m+p}) = Y,\ x: \{\!|l_i:B_i^{\,i\in 1..n+m+p}, X|\!\}$. For such a term, both the composition and the direct coercion return $\mathrm{pack}\ X'::\uparrow(l_i^{\,i\in 1..n}) = (l_i:B_i^{\,i\in n+1..n+m+p}, Y),\ x: \{\!|l_i:B_i^{\,i\in 1..n}, X'|\!\}$. (It is a simple calculation.)

If C is Top, then we simply check that $\lambda(x:A^*)\langle\rangle$ equals $(\lambda(x:A^*)a\ ;\ \lambda(x:B^*)\langle\rangle)$, where a could be any term. This holds by the η rule for records: Top* is $\langle\rangle$, and any x in $\langle\rangle$ equals the empty record $\langle\rangle$.

(Sub Top) Trivial by definition of the coercions.
(Sub Object) Trivial by definition of the coercions.
□

In light of this theorem and its proof, we can write $(E \vdash A<:B)^*$ for the unique coercion obtained from proofs of $E \vdash A<:B$. In fact we omit E, both for brevity and because the coercion is independent of E, though we will not exploit this here.

Lemma 10.3-1 (Coherence)
If f and g are the translations of a term, and they are defined from two $\mathbf{Ob}_{1<:}$ derivations that end in the judgments $E \vdash a:A$ and $E \vdash a:B$, then there exists a derivation of $E \vdash a:C$, with translation h, such that $E \vdash C<:A$, $E \vdash C<:B$, and $f = (C<:A)^*(h)$ and $g = (C<:B)^*(h)$.

Proof
The proof is by induction on derivations. If the last step of either derivation is a subsumption, the result follows from the induction hypothesis, since $(A'<:B')^*$; $(B'<:C')^* = (A'<:C')^*$, as checked in the proof of Theorem 10.3-1, in the case for (Sub Trans). So we can assume that neither derivation ends with an application of (Val Subsumption). Given the form of the system, both derivations must end with the same rule:
(Val x) Given the environment, A and B are determined, and equal; and f and g are immediately identical; we take $C = A = B$ and $h = f = g$.
(Val Object) The form of a determines that A and B both equal the type of the ς-bound variables in a. In this case, the translation is fixed, and f and g are identical; we take $C = A = B$ and $h = f = g$.
(Val Select) Here a is of the form b.m. In one derivation, we must have $E \vdash b:[m:A, m_i:A_i \ ^{i \in I}]$ with a translation f_1 and in the other $E \vdash b:[m:B, m_j:B_j \ ^{j \in J}]$ with a translation g_1. By induction hypothesis, we can derive $E \vdash b:C_1$, for some C_1 subtype of both $[m:A, m_i:A_i \ ^{i \in I}]$ and $[m:B, m_j:B_j \ ^{j \in J}]$. It follows that A and B are equal, for otherwise C_1 cannot exist. In addition, C_1 must have both the forms $[m:A, m_i:A_i \ ^{i \in I}, ...]$ and $[m:B, m_j:B_j \ ^{j \in J}, ...]$. Call h_1 the translation of the derivation of $E \vdash b:C_1$. Then $f_1 = (C_1<:[m:A, m_i:A_i \ ^{i \in I}])^*(h_1)$ and $g_1 = (C_1<:[m:B, m_j:B_j \ ^{j \in J}])^*(h_1)$, by induction hypothesis. Let h be the translation of b.m using the derivation of $E \vdash b:C_1$ plus an application of (Val Select), and let $C = A = B$. It can be shown that $h = f = g$ by calculation.
(Val Override) Analogous to (Val Object).
□

Taking A=B, we obtain:

Theorem 10.3-2 (Coherence)
If f and g are the translations of two $\mathbf{Ob}_{1<:}$ derivations that end with the same value judgment, then f and g are provably equal in the target calculus.

The translation easily extends to the full $\mathbf{FOb}_{<:\mu}$. However, coherence for $\mathbf{FOb}_{<:\mu}$ seems to require new insights. The coherence problems for arrows and universal quantifiers have been essentially solved [Breazu-Tannen, *et al.* 1991]. We do not expect surprises from existential quantifiers. Further work is needed on recursion [Breazu-Tannen, Gunter, Scedrov 1990].

11. Conclusions

Instead of reducing objects to more primitive concepts, we have tried to capture the expected properties of objects, studying typing rules and equational theories. We have obtained an integration of

subsumption and method override that has eluded formalization attempts based on encodings. We have developed an expressive object notation, and in particular we have found that λ-calculi can be encoded in our object calculi.

One of the contributions of this paper is an equational theory of objects that accounts for subsumption and method override. Although the equational theory is simple and possibly incomplete, no such theory seems to exist in the literature. Much remains to be done in finding proof rules for reasoning about objects.

11.1 Related Work

We finish with some comparisons with the most closely related works.

- As we mentioned in section 9, many of the ideas in our treatment of objects originated in the study of Baby Modula-3. That language resembles $\mathbf{FOb}_{1<:\mu}$ in power, although the syntactic details of the two languages are incomparable. For example, Baby Modula-3 includes a limited form of object extension, and its operational semantics has a rather strict evaluation strategy. Its denotational semantics is based on ideals rather than on pers. A per semantics is also briefly sketched in [Abadi 1994], but it is relatively uninteresting, as Baby Modula-3 includes no equational rules.

- Our paper is also closely related in spirit, if not in detail, to [Mitchell, Honsell, Fisher 1993]. We take the same approach of defining an untyped object calculus with override, based on self-application semantics, and then looking for relevant type systems.

  The most significant difference in outcome is that we are able to support subtyping and subsumption. In this, we have been helped by basing our calculus on fixed-size objects, instead of open-ended extensible objects. Without extensible objects, we simulate inheritance by reusing parametrically defined methods. On the other hand, Mitchell *et al.* study extensible objects, providing a more direct modeling of method inheritance, but without support for subtyping. It is not clear whether the good points of the two approaches can be merged.

  Other differences between the two papers are more technical. In their work, Mitchell *et al.* provide an elaborate axiomatization of objects types with Self, while we axiomatize simpler object types, and then obtain Self by second-order encodings. They use a kind system (type operators and row-types) for object extensibility, but need no second-order quantifiers since their Self is built-in. We use quantifiers for Self, but we were able to avoid the complexity of kind systems since our objects are of fixed size. They show soundness by subject reduction, while we provide a model. Therefore, we can justify an equational theory and we can investigate language extensions more easily. Moreover, we have worked out a second-order calculus of objects.

  Many common examples can be expressed in both systems, with some noticeable exceptions. (1) We cannot inherit directly from existing objects, for example producing a color point from an existing point (but see sections 4.3, 7.3, and 8.2). (2) Our numerals, based on a Scott-style encoding, are simpler than theirs, based on a Church-style encoding. We are able to give typings for predecessor and test for zero; they would need universal quantifiers to adopt our typings.

- The TOOPL language [Bruce 1993] has built-in objects with Self, and supports a form of subsumption that is obtained via two distinct subtype relations. TOOPL distinguishes between objects and classes, and provides special syntax for classes, super, and other constructs. Correspondingly, the TOOPL semantics is based on F-bounded quantification and distinguishes between objects and object generators. Methods can be overridden in classes but not in objects.

  The rules for overriding methods are remarkably similar in TOOPL and in our calculi. In both cases the treatment of Self is complex, but our rules are derivable from those for elementary objects without Self. Hence, we may claim some success in explaining rules for Self, short of computing their

denotational meaning. In presence of polymorphism a built-in Self type may lead to rules stronger than those we have derived, like the structural rules of section 8.3.2.

- Our use of existential types is weakly related to that of Pierce and Turner [Pierce, Turner 1994]. We do not use existentials to hide the internal representation of objects; our internal representation is, recursively, a subtype of the object type itself. In their paper, they allow updates to the internal representation of objects, but do not allow method override.

- It is common to encode existential quantifiers from universal quantifiers. However, to our knowledge, the detailed cuper semantics of existentials as unions had not been worked out. Nor had the equational theory of bounded existential quantifiers. Cardone's thesis describes an instructive attempt [Cardone 1990].

- For some time, one of us (L.C.) has searched for a satisfactory encoding of typed objects in terms of typed records. In absence of such encodings, various authors have defined and used rich calculi with records (e.g., [Wand 1989; Harper, Pierce 1990; Cardelli, Mitchell 1991; Pierce, Turner 1994]). A general encoding was not proposed, but many object-flavored examples could be expressed and examined. Mitchell [Mitchell 1990] proposed an encoding that does not respect subtyping. Pierce and Turner [Pierce, Turner 1994] sketched an encoding for the class semantics that does not extend to proper method override.

11.2 Acknowledgments

John Mitchell was helpful during initial discussions about the subject of this paper. John Lamping prompted us to think about encoding covariant components, and also suggested the use of disjoint sums for typing object-oriented numerals. Gordon Plotkin suggested we should have a system with the Self quantifier as the only second-order construct.

Appendix A: Simple-Objects Fragments

These are the typing and equational rules for simple objects.

Δ_{Ob}

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i:B_i\ ^{i\in 1..n}]} \quad (l_i \text{ distinct})$$

(Sub Object)

$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i:B_i\ ^{i\in 1..n+m}] <: [l_i:B_i\ ^{i\in 1..n}]} \quad (l_i \text{ distinct})$$

(Val Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i:B_i\ ^{i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b : A}$$

$\Delta_{=Ob} \cup \Delta_{=Ob<:}$

(Eq Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \leftrightarrow [l_i=\varsigma(x_i:A)b_i'\ ^{i\in 1..n}] : A}$$

(Eq Sub Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $A' \equiv [l_i:B_i\ ^{i\in 1..n}, l_j:B_j\ ^{j\in n+1..m}]$)

$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j:A' \vdash b_j : B_j \quad \forall j \in n+1..m}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \leftrightarrow [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}] : A}$$

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i:B_i\ ^{i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \leftrightarrow a'.l_j \Leftarrow \varsigma(x:A)b' : A}$$

(Eval Select)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \leftrightarrow [l_j=\varsigma(x:A')b, l_i=\varsigma(x_i:A')b_i\ ^{i\in(1..n+m)-\{j\}}] : A}$$

Appendix B: ς-Objects Fragments (Derived Rules)

These are derived rules for the combination of simple objects and the Self quantifier.

$\Delta_{\varsigma+}$

(Type ςObject)

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i \in 1..n}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]}$$

(Sub ςObject)

$$\frac{E,X \vdash B_i\{X^+\} \quad \forall i \in 1..n+m}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}] <: \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]} \quad (l_i \text{ distinct})$$

(Val ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : A(C)}{E \vdash \varsigma(Y<:A=C)\, b\{Y\} : A}$$

(Val ςSelect)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j : B_j\{A^+\}}$$

(Val ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a : A \quad E,\ Y<:A,\ y:A(Y),\ x:A(Y) \vdash b : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A,\ y:A(Y),\ x:A(Y))b : A}$$

$\Delta_{=\varsigma+}$

(Eq ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

$$\frac{E \vdash C <: A' \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : A'(C)}{E \vdash \varsigma(Y<:A=C)b\{Y\} \ \leftrightarrow\ \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Sub ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n+m}]$)

$$\frac{E \vdash C <: A' \quad E,\ x_i:A(C) \vdash b_i\{C\} : B_i\{C\} \ \ \forall i \in 1..n \quad E,\ x_j:A'(C) \vdash b_j\{C\} : B_j\{C\} \ \ \forall j \in n+1..m}{E \vdash \varsigma(Y<:A=C)[l_i=\varsigma(x_i:A(Y))b_i\{Y\}^{\ i\in 1..n}] \ \leftrightarrow\ \varsigma(Y<:A'=C)[l_i=\varsigma(x_i:A'(Y))b_i\{Y\}^{\ i\in 1..n+m}] : A}$$

(Eq ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad j \in 1..n}{E \vdash a.l_j \ \leftrightarrow\ a'.l_j : B_j\{A^+\}}$$

(Eq ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E,\ Y<:A,\ y:A(Y),\ x:A(Y) \vdash b \leftrightarrow b' : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A,\ y:A(Y),\ x:A(Y))b \ \leftrightarrow\ a'.l_j \Leftarrow \varsigma(Y<:A,\ y:A(Y),\ x:A(Y))b' : A}$$

(Eval ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \quad j \in 1..n}{E \vdash c.l_j \ \leftrightarrow\ a\{C\}.l_j : B_j\{A^+\}}$$

(Eval ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\ i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \quad E,\ Y<:A,\ y:A(Y),\ x:A(Y) \vdash b\{Y,y,x\} : B_j\{Y^+\} \quad j \in 1..n}{E \vdash c.l_j \Leftarrow \varsigma(Y<:A,y:A(Y),x:A(Y))b\{Y,y,x\} \ \leftrightarrow\ \varsigma(Z<:A=C)\ a\{Z\}.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,a\{Z\},x\} : A}$$

Appendix C: The $F_{<:\mu}$ Calculus

Environments	$E ::= \emptyset \mid E,x:A \mid E,X<:A$
Type Variables	X,Y
Types	$A,B,C,D ::= X \mid Top \mid A \to B \mid \forall(X<:A)B \mid \exists(X<:A)B \mid \mu(X)A$
Variables	x,y
Values	$a,b,c,d ::= x \mid \lambda(x:A)b \mid b(a) \mid \lambda(X<:A)b \mid b(A)$ $\mid$ pack X<:A=C, b{X}:B{X} $\mid$ open c as X<:A,x:B in d:D $\mid$ fold(μ(X)A, a) $\mid$ unfold(a) $\mid \mu(x:A)a$

This is the standard second-order calculus with subtyping and recursion (with primitive existential quantification). The fragments forming this calculus have been rearranged; the rules are grouped by judgments.

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \quad x \notin dom(E)}{E,x:A \vdash \diamond}$$

(Env X<:)
$$\frac{E \vdash A \quad X \notin dom(E)}{E,X<:A \vdash \diamond}$$

(Type X<:)
$$\frac{E',X<:A,E'' \vdash \diamond}{E',X<:A,E'' \vdash X}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Type Arrow)
$$\frac{E \vdash A \quad E \vdash B}{E \vdash A \to B}$$

(Type All<:)
$$\frac{E,X<:A \vdash B}{E \vdash \forall(X<:A)B}$$

(Type Exists<:)
$$\frac{E,X<:A \vdash B}{E \vdash \exists(X<:A)B}$$

(Type Rec<:)
$$\frac{E,X \vdash A \quad A > X}{E \vdash \mu(X)A}$$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Sub X)
$$\frac{E',X<:A,E'' \vdash \diamond}{E',X<:A,E'' \vdash X<:A}$$

(Sub Arrow)
$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \to B <: A' \to B'}$$

(Sub All)
$$\frac{E \vdash A' <: A \quad E,X<:A' \vdash B <: B'}{E \vdash \forall(X<:A)B <: \forall(X<:A')B'}$$

(Sub Exists)
$$\frac{E \vdash A <: A' \quad E,X<:A \vdash B <: B'}{E \vdash \exists(X<:A)B <: \exists(X<:A')B'}$$

(Sub Rec)
$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y,X<:Y \vdash A<:B}{E \vdash \mu(X)A <: \mu(Y)B}$$

(Val Subsumption)
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

(Val x)
$$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x:A}$$

(Val Fun)
$$\frac{E, x:A \vdash b : B}{E \vdash \lambda(x:A)b : A \to B}$$

(Val Appl)
$$\frac{E \vdash b : A \to B \quad E \vdash a : A}{E \vdash b(a) : B}$$

(Val Fun2<:)
$$\frac{E, X<:A \vdash b : B}{E \vdash \lambda(X<:A)b : \forall(X<:A)B}$$

(Val Appl2<:)
$$\frac{E \vdash b : \forall(X<:A)B\{X\} \quad E \vdash A'<:A}{E \vdash b(A') : B\{A'\}}$$

(Val Pack<:)
$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash (\text{pack } X<:A=C, b\{X\}:B\{X\}) : \exists(X<:A)B\{X\}}$$

(Val Open<:)
$$\frac{E \vdash c : \exists(X<:A)B \quad E \vdash D \quad E, X<:A, x:B \vdash d : D}{E \vdash (\text{open } c \text{ as } X<:A, x:B \text{ in } d:D) : D}$$

(Val Fold)
$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)
$$\frac{E \vdash a : \mu(X)A}{E \vdash \text{unfold}(a) : A\{X \leftarrow \mu(X)A\}}$$

(Val Rec)
$$\frac{E, x:A \vdash a : A}{E \vdash \mu(x:A)a : A}$$

(Eq Symm)
$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)
$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

(Eq Subsumption)
$$\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Top)
$$\frac{E \vdash a:A \quad E \vdash b:B}{E \vdash a \leftrightarrow b : \text{Top}}$$

(Eq x)
$$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x \leftrightarrow x : A}$$

(Eq Fun)
$$\frac{E, x:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \to B}$$

(Eq Appl)
$$\frac{E \vdash b \leftrightarrow b' : A \to B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eq Fun2<:)
$$\frac{E, X<:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X<:A)b \leftrightarrow \lambda(X<:A)b' : \forall(X<:A)B}$$

(Eq Appl2<:)
$$\frac{E \vdash b \leftrightarrow b' : \forall(X<:A)B\{X\} \quad E \vdash A'<:A}{E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}}$$

(Eq Pack<:)
$$\frac{E \vdash C <: A' \quad E \vdash A'<:A \quad E, X<:A' \vdash B'\{X\}<:B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash (\text{pack } X<:A=C, b\{X\}:B\{X\}) \leftrightarrow (\text{pack } X<:A'=C, b'\{X\}:B'\{X\}) : \exists(X<:A)B\{X\}}$$

(Eq Open<:)
$$\frac{E \vdash c \leftrightarrow c' : \exists(X<:A)B \quad E \vdash D \quad E, X<:A, x:B \vdash d \leftrightarrow d' : D}{E \vdash (\text{open } c \text{ as } X<:A, x:B \text{ in } d:D) \leftrightarrow (\text{open } c' \text{ as } X<:A, x:B \text{ in } d':D) : D}$$

(Eq Fold<:)

$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y,X<:Y \vdash A<:B \quad E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) \leftrightarrow \text{fold}(\mu(Y)B, a') : \mu(Y)B}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash \text{unfold}(a) \leftrightarrow \text{unfold}(a') : A\{X \leftarrow \mu(X)A\}}$$

(Eq Rec)

$$\frac{E,x:A \vdash a \leftrightarrow a' : A}{E \vdash \mu(x:A)a \leftrightarrow \mu(x:A)a' : A}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x:A)b : A \rightarrow B \quad E \vdash a : A}{E \vdash (\lambda(x:A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)}{E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B}$$

(Eval Beta2<:)

$$\frac{E \vdash \lambda(X<:A)b : \forall(X<:A)B \quad E \vdash C <: A}{E \vdash (\lambda(X<:A)b)(C) \leftrightarrow b\{X \leftarrow C\} : B\{X \leftarrow C\}}$$

(Eval Eta2<:)

$$\frac{E \vdash b : \forall(X<:A)B \quad X \notin \text{dom}(E)}{E \vdash \lambda(X<:A)b(X) \leftrightarrow b : \forall(X<:A)B}$$

(Eval Unpack<:) (where $c \equiv \text{pack } X<:A=C, b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X<:A)B\{X\} \quad E \vdash D \quad E,X<:A,x:B\{X\} \vdash d\{X,x\} : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B\{X\} \text{ in } d\{X,x\}:D) \leftrightarrow d\{C,b\{C\}\} : D}$$

(Eval Repack<:)

$$\frac{E \vdash b : \exists(X<:A)B\{X\} \quad E,y:\exists(X<:A)B\{X\} \vdash d\{y\} : D}{E \vdash (\text{open } b \text{ as } X<:A,x:B\{X\} \text{ in } d\{\text{pack } X'<:A=X, x:B\{X'\}\}:D) \leftrightarrow d\{b\} : D}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash \text{fold}(\mu(X)A,\text{unfold}(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{unfold}(\text{fold}(\mu(X)A,a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}}$$

(Eval Rec)

$$\frac{E,x:A \vdash a : A}{E \vdash \mu(x:A)a \leftrightarrow a\{x \leftarrow \mu(x:A)a\} : A}$$

Appendix D: The ςOb Calculus

ςOb is our minimal second-order object calculus. It is obtained by combining the rules for object types (appendix A) with the Self quantifier (section 6.5) taken as a primitive, plus some general rules from $F_{<:}$ (appendix C). The rules for ς-objects (appendix B) are derivable from the ones shown here. The relation A≻X is defined as in appendix E, with in addition ς(X)B ≻ Y if B≻Y.

$$A,B ::= X \mid Top \mid [l_i:B_i\ ^{i\in 1..n}] \mid \varsigma(X)B$$
$$a,b ::= x \mid [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \mid a.l \mid a.l \Leftarrow \varsigma(x:A)b \mid \varsigma(X<:A=B)b \mid \text{use a as } X<:A,\ y:B \text{ in } b:D$$

(Env ∅) $\dfrac{}{\emptyset \vdash \diamond}$ (Env x) $\dfrac{E \vdash A \quad x \notin dom(E)}{E,x:A \vdash \diamond}$ (Env X<:) $\dfrac{E \vdash A \quad X \notin dom(E)}{E,X<:A \vdash \diamond}$

(Type X<:) $\dfrac{E',X<:A,E'' \vdash \diamond}{E',X<:A,E'' \vdash X}$ (Type Top) $\dfrac{E \vdash \diamond}{E \vdash Top}$ (Type Object) $\dfrac{E \vdash B_i \quad \forall i\in 1..n \ (l_i \text{ distinct})}{E \vdash [l_i:B_i\ ^{i\in 1..n}]}$ (Type Self) $\dfrac{E,X<:Top \vdash B \quad B\succ X}{E \vdash \varsigma(X)B}$

(Sub Refl) $\dfrac{E \vdash A}{E \vdash A <: A}$ (Sub Trans) $\dfrac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$ (Sub X) $\dfrac{E',X<:A,E'' \vdash \diamond}{E',X<:A,E'' \vdash X<:A}$

(Sub Top) $\dfrac{E \vdash A}{E \vdash A <: Top}$ (Sub Object) $\dfrac{E \vdash B_i \quad \forall i\in 1..n+m \quad (l_i \text{ distinct})}{E \vdash [l_i:B_i\ ^{i\in 1..n+m}] <: [l_i:B_i\ ^{i\in 1..n}]}$ (Sub Self) $\dfrac{E,X<:Top \vdash B <: B' \quad B,B'\succ X}{E \vdash \varsigma(X)B <: \varsigma(X)B'}$

(Val Subsumption) $\dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$ (Val x) $\dfrac{E',x:A,E'' \vdash \diamond}{E',x:A,E'' \vdash x:A}$ (Val Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$) $\dfrac{E,\ x_i:A \vdash b_i : B_i \quad \forall i\in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] : A}$

(Val Select) $\dfrac{E \vdash a : [l_i:B_i\ ^{i\in 1..n}] \quad j\in 1..n}{E \vdash a.l_j : B_j}$ (Val Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$) $\dfrac{E \vdash a : A \quad E,\ x:A \vdash b : B_j \quad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b : A}$

(Val Self) (where $A \equiv \varsigma(X)B\{X\}$) $\dfrac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash \varsigma(Y<:A=C)\ b\{Y\} : A}$ (Val Use) (where $A \equiv \varsigma(X)B\{X\}$) $\dfrac{E \vdash c : A \quad E \vdash D \quad E,\ Y<:A,\ y:B\{Y\} \vdash d : D}{E \vdash (\text{use c as } Y<:A,\ y:B\{Y\} \text{ in d:D}) : D}$

(Eq Symm)

$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

(Eq Subsumption)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x \leftrightarrow x : A}$$

(Eq Top)

$$\frac{E \vdash a{:}A \quad E \vdash b{:}B}{E \vdash a \leftrightarrow b : Top}$$

(Eq Object) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i{:}A)b_i\ ^{i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i{:}A)b_i'\ ^{i \in 1..n}] : A}$$

(Eq Sub Object) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $A' \equiv [l_i{:}B_i\ ^{i \in 1..n}, l_j{:}B_j\ ^{j \in n+1..m}]$)

$$\frac{E, x_i{:}A \vdash b_i : B_i \quad \forall i \in 1..n \qquad E, x_j{:}A' \vdash b_j : B_j \quad \forall j \in n+1..m}{E \vdash [l_i = \varsigma(x_i{:}A)b_i\ ^{i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i{:}A')b_i\ ^{i \in 1..n+m}] : A}$$

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i{:}B_i\ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, x{:}A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b \leftrightarrow a'.l_j \Leftarrow \varsigma(x{:}A)b' : A}$$

(Eq Self) (where $A \equiv \varsigma(X)B\{X\}$, $A' \equiv \varsigma(X)B'\{X\}$)

$$\frac{E \vdash C <: A' \quad E, X \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} \leftrightarrow \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c \leftrightarrow c' : A \quad E \vdash D \quad E, Y<:A, y{:}B\{Y\} \vdash d \leftrightarrow d' : D}{E \vdash (use\ c\ as\ Y<:A, y{:}B\{Y\}\ in\ d{:}D) \leftrightarrow (use\ c'\ as\ Y<:A, y{:}B\{Y\}\ in\ d'{:}D) : D}$$

(Eval Select) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i{:}A')b_i\ ^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i{:}A')b_i\ ^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, x{:}A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b \leftrightarrow [l_j = \varsigma(x{:}A')b, l_i = \varsigma(x_i{:}A')b_i\ ^{i \in (1..n+m)-\{j\}}] : A}$$

(Eval Unself) (where $A \equiv \varsigma(X)B\{X\}$, $c \equiv \varsigma(Z<:A=C)b\{Z\}$)

$$\frac{E \vdash c : A \quad E \vdash D \quad E, Y<:A, y{:}B\{Y\} \vdash d\{Y,y\} : D}{E \vdash (use\ c\ as\ Y<:A, y{:}B\{Y\}\ in\ d\{Y,y\}{:}D) \leftrightarrow d\{C, b\{C\}\} : D}$$

(Eval Reself) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash b : A \quad E, y{:}A \vdash d\{y\} : D}{E \vdash (use\ b\ as\ Y<:A, y{:}B\{Y\}\ in\ d\{\varsigma(Y'<:A=Y)y\} :A) \leftrightarrow d\{b\} : D}$$

Appendix E: Formal Contractiveness for FOb$_{<:\mu}$

The relation $A \succ Y$ (type expression A is formally contractive in variable Y) is defined as follows:

$X \succ Y$	if $X \neq Y$	
$Top \succ Y$	always	
$[l_i:B_i\ ^{i \in 1..n}] \succ Y$	always	
$A \rightarrow B \succ Y$	always	
$\mu(X)A \succ Y$	if $A \succ Y$	
$\forall(X<:A)B \succ Y$	if $B \succ X$ and $B \succ Y$	(no requirement on A)
$\exists(X<:A)B \succ Y$	if $B \succ X$ and $B \succ Y$	(no requirement on A)

As usual, type expressions are identified up to renaming of bound variables.

Appendix F: The Target Calculus

Environments	$E ::= \emptyset \mid E,x:A \mid E,X::K$
Kinds	$K ::= Type \mid \uparrow(l_1..l_n)$
Type/Row Variables	X,Y
Types	$A,B,C,D ::= X \mid \langle R\rangle \mid A\to B \mid \forall(X::K)B \mid \exists(X::K)B \mid \mu(X)A$
Rows	$R ::= X \mid \emptyset \mid l:B,R$
Variables	x,y
Values	$a,b,c,d ::= x \mid \langle l_i=b_i\ ^{i\in 1..n}\rangle \mid a\cdot l \mid a\cdot l:=b$
	$\mid \lambda(x:A)b \mid b(a) \mid \lambda(X::K)b \mid b(A)$
	$\mid$ pack X::K=C, b{X}:B{X} $\mid$ open c as X::K,x:B in d:D
	$\mid$ fold(μ(X)A, a) $\mid$ unfold(a) $\mid \mu(x:A)a$

E,X,E' stands for E,X::Type,E'.
$E \vdash A$ stands for $E \vdash A::Type$.
$(l_1..l_n)$ is a set of labels; when we write $(l,l_1..l_n)$ we imply that $l \neq l_1..l_n$.

(Env $\emptyset$) $\dfrac{}{\emptyset \vdash \diamond}$ (Env x) $\dfrac{E \vdash A \quad x\notin dom(E)}{E,x:A \vdash \diamond}$ (Env X) $\dfrac{E \vdash K\ kind \quad X\notin dom(E)}{E,X::K \vdash \diamond}$

(Kind Type) $\dfrac{E \vdash \diamond}{E \vdash Type\ kind}$ (Kind Row) $\dfrac{E \vdash \diamond}{E \vdash \uparrow(l_1..l_n)\ kind}$

(Type X) $\dfrac{E',X,E'' \vdash \diamond}{E',X,E'' \vdash X}$ (Type Arrow) $\dfrac{E \vdash A \quad E \vdash B}{E \vdash A\to B}$ (Type Record) $\dfrac{E \vdash R::\uparrow()}{E \vdash \langle R\rangle}$

(Type All) $\dfrac{E,X::K \vdash B}{E \vdash \forall(X::K)B}$ (Type Exists) $\dfrac{E,X::K \vdash B}{E \vdash \exists(X::K)B}$ (Type Rec) $\dfrac{E,X \vdash A \quad A\succ X}{E \vdash \mu(X)A}$

(Row X) $\dfrac{E',X::\uparrow(l_1..l_n),E'' \vdash \diamond}{E',X::\uparrow(l_1..l_n),E'' \vdash X::\uparrow(l_1..l_n)}$ (Row $\emptyset$) $\dfrac{E \vdash \diamond}{E \vdash \emptyset :: \uparrow(l_1..l_n)}$ (Row Cons) $\dfrac{E \vdash R::\uparrow(l,l_1..l_n) \quad E \vdash B}{E \vdash l:B,R :: \uparrow(l_1..l_n)}$

(Val x)
$$\frac{E',x:A,E'' \vdash \diamond}{E',x:A,E'' \vdash x:A}$$

(Val Fun)
$$\frac{E,x:A \vdash b : B}{E \vdash \lambda(x:A)b : A \to B}$$

(Val Appl)
$$\frac{E \vdash b : A \to B \quad E \vdash a : A}{E \vdash b(a) : B}$$

(Val Record)
$$\frac{E \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash \langle l_i = b_{i\ i\in 1..n}\rangle : \langle l_i : B_{i\ i\in 1..n}, \emptyset\rangle}$$

(Val Select)
$$\frac{E \vdash a : \langle l:B,R\rangle}{E \vdash a\cdot l : B}$$

(Val Update)
$$\frac{E \vdash a : \langle l:B,R\rangle \quad E \vdash b : B}{E \vdash a\cdot l := b : \langle l:B,R\rangle}$$

(Val Fun2)
$$\frac{E,X::K \vdash b : B}{E \vdash \lambda(X::K)b : \forall(X::K)B}$$

(Val Appl2)
$$\frac{E \vdash b : \forall(X::K)B\{X\} \quad E \vdash A::K}{E \vdash b(A) : B\{A\}}$$

(Val Pack)
$$\frac{E \vdash A::K \quad E \vdash b\{A\} : B\{A\}}{E \vdash (\text{pack } X::K=A,\ b\{X\}:B\{X\}) : \exists(X::K)B\{X\}}$$

(Val Open)
$$\frac{E \vdash c : \exists(X::K)B \quad E \vdash D \quad E,X::K,x:B \vdash d : D}{E \vdash (\text{open } c \text{ as } X::K,x:B \text{ in } d:D) : D}$$

(Val Fold)
$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)
$$\frac{E \vdash a : \mu(X)A}{E \vdash \text{unfold}(a) : A\{X \leftarrow \mu(X)A\}}$$

(Val Rec)
$$\frac{E,x:A \vdash a : A}{E \vdash \mu(x:A)a : A}$$

(Eq Symm)
$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)
$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

(Eq x)
$$\frac{E',x:A,E'' \vdash \diamond}{E',x:A,E'' \vdash x \leftrightarrow x : A}$$

(Eq Fun)
$$\frac{E,x:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \to B}$$

(Eq Appl)
$$\frac{E \vdash b \leftrightarrow b' : A \to B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eq Record)
$$\frac{E \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash \langle l_i = b_{i\ i\in 1..n}\rangle \leftrightarrow \langle l_i = b_i'\ {}_{i\in 1..n}\rangle : \langle l_i : B_{i\ i\in 1..n}, \emptyset\rangle}$$

(Eq Select)
$$\frac{E \vdash a \leftrightarrow a' : \langle l:B,R\rangle}{E \vdash a\cdot l \leftrightarrow a'\cdot l : B}$$

(Eq Update)
$$\frac{E \vdash a \leftrightarrow a' : \langle l:B,R\rangle \quad E \vdash b \leftrightarrow b' : B}{E \vdash a\cdot l := b \leftrightarrow a'\cdot l := b' : \langle l:B,R\rangle}$$

(Eq Fun2)
$$\frac{E,X::K \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X::K)b \leftrightarrow \lambda(X::K)b' : \forall(X::K)B}$$

(Eq Appl2)
$$\frac{E \vdash b \leftrightarrow b' : \forall(X::K)B\{X\} \quad E \vdash A::K}{E \vdash b(A) \leftrightarrow b'(A) : B\{A\}}$$

(Eq Pack)

$$\frac{E \vdash A::K \quad E,X::K \vdash B\{X\} \quad E \vdash b\{A\} \leftrightarrow b'\{A\} : B\{A\}}{E \vdash (\text{pack } X::K=A, b\{X\}:B\{X\}) \leftrightarrow (\text{pack } X::K=A, b'\{X\}:B\{X\}) : \exists(X::K)B\{X\}}$$

(Eq Open)

$$\frac{E \vdash c \leftrightarrow c' : \exists(X::K)B \quad E \vdash D \quad E,X::K,x:B \vdash d \leftrightarrow d' : D}{E \vdash (\text{open } c \text{ as } X::K,x:B \text{ in } d:D) \leftrightarrow (\text{open } c' \text{ as } X::K,x:B \text{ in } d':D) : D}$$

(Eq Fold)

$$\frac{E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) \leftrightarrow \text{fold}(\mu(Y)A, a') : \mu(Y)A}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash \text{unfold}(a) \leftrightarrow \text{unfold}(a') : A\{X \leftarrow \mu(X)A\}}$$

(Eq Rec)

$$\frac{E,x:A \vdash a \leftrightarrow a' : A}{E \vdash \mu(x:A)a \leftrightarrow \mu(x:A)a' : A}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x:A)b : A \to B \quad E \vdash a : A}{E \vdash (\lambda(x:A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \to B \quad x \notin \text{dom}(E)}{E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \to B}$$

(Eval Select)

$$\frac{E \vdash a : \langle l:B, l_i:B_i \ ^{i \in 1..n}, \emptyset \rangle}{E \vdash a \cdot l \leftrightarrow b : B}$$

(Eval Update) (where $a \equiv \langle l=b, l_i=b_i \ ^{i \in 1..n} \rangle$)

$$\frac{E \vdash a : \langle l:B, l_i:B_i \ ^{i \in 1..n}, \emptyset \rangle \quad E \vdash b' : B}{E \vdash a \cdot l:=b' \leftrightarrow \langle l=b', l_i=b_i \ ^{i \in 1..n} \rangle : \langle l:B, l_i:B_i \ ^{i \in 1..n}, \emptyset \rangle}$$

(Eval Rerecord)

$$\frac{E \vdash a : \langle l_i:B_i \ ^{i \in 1..n}, \emptyset \rangle}{E \vdash a \leftrightarrow \langle l_i=a \cdot l_i \ ^{i \in 1..n} \rangle : \langle l_i:B_i \ ^{i \in 1..n}, \emptyset \rangle}$$

(Eval Beta2)

$$\frac{E \vdash \lambda(X::K)b : \forall(X::K)B \quad E \vdash A::K}{E \vdash (\lambda(X::K)b)(A) \leftrightarrow b\{X \leftarrow A\} : B\{X \leftarrow A\}}$$

(Eval Eta2)

$$\frac{E \vdash b : \forall(X::K)B \quad X \notin \text{dom}(E)}{E \vdash \lambda(X::K)b(X) \leftrightarrow b : \forall(X::K)B}$$

(Eval Unpack) (where $c \equiv \text{pack } X::K=A, b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X::K)B\{X\} \quad E \vdash D \quad E,X::K,x:B\{X\} \vdash d\{X,x\} : D}{E \vdash (\text{open } c \text{ as } X::K,x:B\{X\} \text{ in } d\{X,x\}:D) \leftrightarrow d\{A,b\{A\}\} : D}$$

(Eval Repack)

$$\frac{E \vdash b : \exists(X::K)B\{X\} \quad E,y:\exists(X::K)B\{X\} \vdash d\{y\} : D}{E \vdash (\text{open } b \text{ as } X::K,x:B\{X\} \text{ in } d\{\text{pack } X'::K=X, x:B\{X'\}\}:D) \leftrightarrow d\{b\} : D}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash \text{fold}(\mu(X)A,\text{unfold}(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash \text{unfold}(\text{fold}(\mu(X)A,a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}}$$

(Eval Rec)

$$\frac{E,x{:}A \vdash a : A}{E \vdash \mu(x{:}A)a \;\leftrightarrow\; a\{x \leftarrow \mu(x{:}A)a\} : A}$$

Formal contractiveness.
The relation $A \succ Y$ (type expression A is formally contractive in variable Y) is defined as follows:

$X \succ Y$	if $X \neq Y$
$\langle R \rangle \succ Y$	always
$A \rightarrow B \succ Y$	always
$\forall(X::K)B \succ Y$	if $B \succ Y$
$\exists(X::K)B \succ Y$	if $B \succ Y$
$\mu(X)A \succ Y$	if $A \succ Y$

References

[Abadi 1994] M. Abadi, **Baby Modula-3 and a theory of objects**. *Journal of Functional Programming* **4**(2).

[Abadi, *et al.* 1991] M. Abadi, L. Cardelli, B. Pierce, and G.D. Plotkin, **Dynamic typing in a statically typed language**. *ACM Transactions on Programming Languages and Systems* **13**(2), 237-268.

[Abadi, Plotkin 1990] M. Abadi and G. Plotkin. **A per model of polymorphism and recursive types**. *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*.

[Aczel, Mendler 1989] P. Aczel and N. Mendler. **A final co-algebra theorem**. *Proc. Category Theory and Computer Science*. Springer-Verlag.

[Amadio 1989] R.M. Amadio. **Recursion over realizability structures**. TR1/89. Dipartimento di Informatica, Università di Pisa.

[Amadio, Cardelli 1991] R.M. Amadio and L. Cardelli. **Subtyping recursive types**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*.

[Barendregt 1985] H.P. Barendregt, **The lambda-calculus, its syntax and semantics**. North-Holland.

[Böhm, Berarducci 1985] C. Böhm and A. Berarducci, **Automatic synthesis of typed λ-programs on term algebras**. *Theoretical Computer Science* **39**, 135-154.

[Breazu-Tannen, *et al.* 1991] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov, **Inheritance and explicit coercion**. *Information and Computation* **93**(1), 172-221.

[Breazu-Tannen, Gunter, Scedrov 1990] V. Breazu-Tannen, C. Gunter, and A. Scedrov. **Computing with coercions**. *Proc. 1990 ACM Conference on Lisp and Functional Programming*.

[Bruce 1993] K. Bruce. **A paradigmatic object-oriented programming language: design, static typing, and semantics**. Technical Report No. CS-92-01, revised (to appear in the Journal of Functional Programming). Williams College.

[Cardelli 1988] L. Cardelli, **A semantics of multiple inheritance**. *Information and Computation* **76**, 138-164.

[Cardelli 1991] L. Cardelli. **Extensible records in a pure calculus of subtyping**. Technical Report n.81. DEC Systems Research Center.

[Cardelli, Mitchell 1991] L. Cardelli and J.C. Mitchell, **Operations on records**. *Mathematical Structures in Computer Science* **1**(1), 3-48.

[Cardelli, *et al.* 1991] L. Cardelli, J.C. Mitchell, S. Martini, and A. Scedrov. **An extension of system F with subtyping**. *Proc. Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science 526. Springer-Verlag.

[Cardelli, Wegner 1985] L. Cardelli and P. Wegner, **On understanding types, data abstraction and polymorphism**. *Computing Surveys* **17**(4), 471-522.

[Cardone 1989] F. Cardone. **Relational semantics for recursive types and bounded quantification**. *Proc. Automata, Languages and Programming*. Lecture Notes in Computer Science 372. Springer-Verlag.

[Cardone 1990] F. Cardone. **Tipi ricorsivi e inheritance in linguaggi funzionali**, Dipartimento di Informatica, Università di Torino.

[Cook, Hill, Canning 1990] W. Cook, W. Hill, and P. Canning. **Inheritance is not subtyping**. *Proc. Seventeenth Annual ACM Symposium on Principles of Programming Languages*.

[Cook 1989a] W.R. Cook. **A denotational semantics of inheritance**. Ph.D. Thesis, Computer Science Dept., Brown University.

[Cook 1989b] W.R. Cook. **A proposal for making Eiffel type-safe**. *Proc. European Conference of Object-Oriented Programming*.

[Curien, Ghelli 1992] P.-L. Curien and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$**. *Mathematical Structures in Computer Science* **2**(1), 55-91.

[Dahl, Nygaard 1966] O. Dahl and K. Nygaard, **Simula, an Algol-based simulation language**. *Communications of the ACM* **9**, 671-678.

[Girard, Lafont, Taylor 1989] J.-Y. Girard, Y. Lafont, and P. Taylor, **Proofs and types**. Cambridge University Press.

[Goldberg, Robson 1983] A. Goldberg and D. Robson, **Smalltalk-80. The language and its implementation**. Addison-Wesley.

[Harper, Pierce 1990] R. Harper and B. Pierce. **A record calculus with symmetric concatenation**. Technical Report CMU-CS-90-157. CMU.

[Hofmann, Pierce 1994] M. Hofmann and B.C. Pierce. **A unifying type-theoretic framework for objects**. *Proc. Symposium on Theoretical Aspects of Computer Science*.

[Kamin 1988] S. Kamin. **Inheritance in Smalltalk-80: a denotational definition**. *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*.

[MacQueen, Plotkin, Sethi 1986] D.B. MacQueen, G.D. Plotkin, and R. Sethi, **An ideal model for recursive polymorphic types**. *Information and Control* **71**, 95-130.

[Meyer 1988] B. Meyer, **Object-oriented software construction**. Prentice Hall.

[Mitchell 1990] J.C. Mitchell. **Toward a typed foundation for method specialization and inheritance**. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*.

[Mitchell, Honsell, Fisher 1993] J.C. Mitchell, F. Honsell, and K. Fisher. **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*.

[Mitchell, Plotkin 1985] J.C. Mitchell and G.D. Plotkin. **Abstract types have existential type**. *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*.

[Nelson 1991] G. Nelson, ed. **Systems programming with Modula-3**. Prentice Hall.

[Palsberg 1993] J. Palsberg, **Efficient inference for object types**. Manuscript.

[Pierce, Turner 1994] B.C. Pierce and D.N. Turner, **Simple type-theoretic foundations for object-oriented programming**. *Journal of Functional Programming* **4**(2).

[Plotkin, Abadi, Cardelli 1993] G.D. Plotkin, M. Abadi, and L. Cardelli, **Subtyping and parametricity**. Manuscript.

[Reynolds 1983] J.C. Reynolds, **Types, abstraction, and parametric polymorphism**. In *Information Processing,* R.E.A. Mason, ed. North Holland. 513-523.

[Smyth, Plotkin 1982] M.B. Smyth and G.D. Plotkin, **The category-theoretic solution of recursive domain equations**. *SIAM Journal of Computing* **11**(4), 761-783.

[Wadsworth 1980] C. Wadsworth, **Some unusual λ-calculus numeral systems**. In *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism,* J.P. Seldin and J.R. Hindley, ed. Academic Press.

[Wand 1987] M. Wand. **Complete type inference for simple objects**. *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science*.

[Wand 1989] M. Wand. **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*.

[Weinreb, Moon 1981] D. Weinreb and D. Moon, **Lisp machine manual, Fourth Edition, Chapter 20: Objects, message passing, and flavors**. Symbolics Inc.