A Semantics of Object Types

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract: We give a semantics for a typed object calculus, an extension of System **F** with object subsumption and method override. We interpret the calculus in a per model, proving the soundness of both typing and equational rules. This semantics suggests a syntactic translation from our calculus into a simpler calculus with neither subtyping nor objects.

1. Objects, Records, and Functions

Despite the many formal accounts of object-oriented languages, the meaning and the properties of object types remain unclear. In particular, the soundness of object subtyping depends on invariants difficult to capture with standard type constructions; attempts based on record types have been inspiring but not compelling.

In order to study object types in a clear setting, we give semantics to an extension of Girard's System **F** [Girard, Lafont, Taylor 1989] with subtyping, recursion, and some basic object constructs. Like all common object-oriented languages, this calculus supports object subsumption and method override. With subsumption, a new object with more methods can replace an old object transparently. Override is the operation that modifies the behavior of an object, or class, by replacing one of its methods. Neither subsumption nor override is too hard to model in isolation, but their combination has been problematic (see Section 6).

Our starting point is the naive view that an object is a record of methods, and that each method is a function. When a method of an object o is invoked, the corresponding function is applied to o. This view of objects as records of functions is often used informally in the literature and it underlies all implementations of standard (single-dispatch) object-oriented languages. In this work, we extend this view to object types.

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 extension of System **F**, with recursion and records, but neither subtyping nor objects.

The rest of this introduction describes objects, their intended behavior, and the semantic problems that our approach is designed to solve. Some of this material is borrowed from [Abadi, Cardelli 1994b; Abadi, Cardelli 1994c], where we develop typed and untyped object calculi. We start by describing an untyped calculus that includes object formation, method invocation, and method override.

An *object* is a collection of components $[l_i{=}a_i \; ^{i\in 1..n}]$, for distinct labels l_i and associated methods a_i. The order of these components does not matter. A *method* is a function having a special parameter, called *self*. A *proper method* makes use of its self parameter; a *field* is a method that ignores self. The letter ς is used as a binder for self, like a special λ binder; $\varsigma(x)b$ is a method with self parameter x and with body b. The object containing a given method is called the method's *host* object.

A *method invocation* (or *selection*) is written $o.l_j$, where l_j is a label of o. It reduces to the result substituting o for the self parameter in the body of the method named l_j. Thus, a method can be applied only to its host object; this invariant is essential for typing objects and for reasoning about them.

A *method override* (or *update*) is written $o.l_j \Leftarrow \varsigma(y)b$. The intent is to replace the method named l_j of o with $\varsigma(y)b$. Our semantics of override is functional: the result of an override is a copy of the object where the overridden method has been replaced by the new one. This form of method override is more general than usual, in that it applies to objects rather than classes; the generality does not complicate our formal treatment and has advantages in simplicity and expressiveness.

We give a direct, informal semantics of objects, viewing them as primitive:

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 $\varsigma(x_i)b_i$		
$o.l_j$	$\leadsto$	$b_j\{x_j{\leftarrow}o\}$	selection/invocation ($j\in 1..n$)
$o.l_j \Leftarrow \varsigma(y)b$	$\leadsto$	$[l_i{=}\varsigma(x_i)b_i \; ^{i\in(1..n)-\{j\}}, l_j{=}\varsigma(y)b]$	update/override ($j\in 1..n$)

Notation We write $\Phi_i \; ^{i\in 1..n}$ for a sequence $\Phi_1,...,\Phi_n$. The substitution of c for the free occurrences of x in b is $b\{x{\leftarrow}c\}$. We use $\leadsto$ for "rewrites to", $\triangleq$ for definitional equality, $\equiv$ for syntactic identity, and = for provable equality between terms. We identify terms up to renaming of bound variables.

While the primitive semantics reflects the programmer's view of objects, the implementations of standard object-oriented languages are based on self-application. In the self-application semantics [Kamin 1988], methods are functions, objects are records, invocation is record selection plus self-application, and override is record update. We use the notation $\langle l_i{=}a_i \; ^{i\in 1..n}\rangle$ for the record with labels l_i and fields a_i; $r{\cdot}l_j$ for record selection (extracting the l_j component of r); and $r{\cdot}l_j{:=}b$ for record 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	$\triangleq$	$\langle l_i{=}\lambda(x_i)b_i \; ^{i\in 1..n}\rangle$		
$o.l_j$	$\triangleq$	$o{\cdot}l_j(o)$	=	$b_j\{x_j{\leftarrow}o\}$
	($j\in 1..n$)			
$o.l_j \Leftarrow \varsigma(y)b$	$\triangleq$	$o{\cdot}l_j{:=}\lambda(y)b$	=	$[l_i{=}\varsigma(x_i)b_i \; ^{i\in(1..n)-\{j\}}, l_j{=}\varsigma(y)b]$
	($j\in 1..n$)			

The provable equalities follow from the usual λ-calculus rules. They show that the self-application semantics matches the primitive semantics. Hence, untyped objects can be faithfully interpreted by λ-abstraction, application, and record constructions. In turn, records can be reduced to pure λ-terms.

Unfortunately, the self-application semantics does not directly extend to typed calculi. 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 ($i\in 1..n$). By mapping ς to λ, 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, such as the inclusion of $[l_1{:}B_1, l_2{:}B_2]$ into $[l_1{:}B_1]$.

Let us consider, for example, the type of polar points, Point $\triangleq [\rho,\theta{:}\ Real]$. A proper method associated with θ could return 0.0 whenever the ρ component is not positive. In a naive self-application semantics, the type Point is interpreted as a record type: the solution of the type equation Point $= \langle \rho,\theta{:}\ Point \rightarrow Real\rangle$. But this type does not include the solution of ColorPoint $= \langle \rho,\theta{:}\ ColorPoint \rightarrow Real,\ c{:}ColorPoint \rightarrow Color\rangle$, which is the interpretation of the type ColorPoint $\triangleq [\rho,\theta{:}\ Real,\ c{:}\ Color]$.

Semantically, we can 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 $X = \langle \rho,\theta{:}\ X \rightarrow Real, \ldots \rangle$, including for example $X = \langle \rho,\theta{:}\ X \rightarrow Real\rangle$ and $X = \langle \rho,\theta{:}\ X \rightarrow Real, c{:}X \rightarrow Color\rangle$. With this definition, ColorPoint is forced to be a subtype of Point. Our denotational semantics is based on the simple idea just described; the details necessary are, as usual, intricate.

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.

The next section reviews our object calculus more formally. Sections 3 and 4 describe a denotational semantics for the calculus. Section 5 concerns the translation into a calculus without objects. We conclude in Section 6 with comparisons with related work. An appendix summarizes our formal systems. Examples and proofs can be found in [Abadi, Cardelli 1994a].

2. A Theory of Primitive Objects

We now review the typed object calculus, leaving most rules for the appendix. 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 $E \vdash \Im$, for an environment E and an assertion $\Im$ depending on the judgment. An antecedent of the form "$E,E_i \vdash \Im_i \quad \forall i\in 1..n$" is an abbreviation for n antecedents "$E,E_1 \vdash \Im_1$ ... $E,E_n \vdash \Im_n$" if n>0, and if n=0 for "$E \vdash \diamond$", which means E is well-formed. Instead, a rule containing "$j\in 1..n$" indicates that there are n separate rules, one for each j. Environments contain typing assumptions for variables; they can also contain type-variable declarations and subtyping assumptions.

2.1 Object Typing and Subtyping

We start with the typing rules for objects. We give rules for proving type judgments $E \vdash B$ ("B is a well-formed type in the environment E") and value judgments $E \vdash b : B$ ("b has type B in the environment E").

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 $B_1,\ldots,B_n$. When writing $[l_i{:}B_i\ ^{i\in 1..n}]$, we always assume that the l_i are distinct and that permutations do not matter. The type $[l_i{:}B_i\ ^{i\in 1..n}]$ exhibits only the result types B_i, and not the types of ς-bound variables. The types of all these variables is $[l_i{:}B_i\ ^{i\in 1..n}]$, so no information is missing. When a method l_i is invoked, it produces a result of type B_i. A method can be overridden while preserving the type of its host object.

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

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

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

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

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods. We call this notion *subsumption*, and say that an object can *subsume* another one. 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 A also belongs to any supertype B of A, and can subsume objects in B. The judgment $E \vdash A <: B$ asserts A is a subtype of B in the environment E.

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

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

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

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

For convenience, we add a constant, Top, a supertype of every type. 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, object types are *invariant* in their components: $[l_i{:}B_i\ ^{i\in 1..n+m}]<:[l_i{:}B_i'\ ^{i\in 1..n}]$ requires $B_i\equiv B_i'$ for $i\in 1..n$. This is necessary for soundness.

The full first-order calculus of objects with subtyping is called $\mathbf{Ob}_{1<:}$ (see the appendix). To facilitate comparison with other first-order calculi, $\mathbf{Ob}_{1<:}$ includes constants and their sorts, but we do not treat them formally in this paper.

2.2 Functions, Recursion, and Quantification

Functions (in the form of λ-terms) and recursive values can be added to $\mathbf{Ob}_{1<:}$ via standard rules, though functions can also be encoded in terms of objects [Abadi, Cardelli 1994c].

In order to add recursive types, we define a syntactic criterion for contractiveness in the sense of [MacQueen, Plotkin, Sethi 1986]. If A is formally contractive in the variable X, then the fixpoint $\mu(X)A$ exists and is unique. Object types are formally contractive in all their variables.

Further, we introduce bounded universal quantifiers and existential quantifiers, obtaining bounded polymorphic functions and partially abstract data types [Cardelli, Wegner 1985]. We invent no new constructions. However, quantifiers can be combined with recursive types to represent interesting notions, such as the *Self* quantifier [Abadi, Cardelli 1994b]. Our starting point for second-order calculi is $\boldsymbol{F}_{<:}$, as described in [Cardelli, *et al.* 1991], but we assume only the simpler equational theory of [Curien, Ghelli 1992]. Within $\boldsymbol{F}_{<:}$ it is possible to encode bounded existential quantifiers. However, we take them as primitive along with an "eta" rule that is not available through the encoding ((Eval Repack <:) in the appendix).

$\mathbf{F}_{<:\mu}$ is the extension of $\boldsymbol{F}_{<:}$ with existentials and recursion. $\mathbf{Ob}_{<:\mu}$ is the second-order calculus of objects with recursion and subtyping; it is $\mathbf{Ob}_{1<:}$ plus quantifiers and recursion. $\mathbf{FOb}_{<:\mu}$ is the extension of $\mathbf{Ob}_{<:\mu}$ with function types. We work in $\mathbf{FOb}_{<:\mu}$, although it can be encoded in $\mathbf{Ob}_{<:\mu}$ [Abadi, Cardelli 1994b]. We do not know whether $\mathbf{Ob}_{<:\mu}$ can be encoded in $\mathbf{F}_{<:\mu}$ while preserving subtypings; Section 5 deals with an encoding that translates subtypings into coercions.

2.3 Equational Theories

We associate an equational theory with each of our calculi. The judgment $E \vdash b \leftrightarrow c : A$ asserts that b and c are equal as elements of A. We give only the main rules for objects and subtyping: three rules motivated by the use of subtyping, and two evaluation rules corresponding to the semantics of the untyped calculus.

(Eq Subsumption)

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

(Eq Top)

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

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

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

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

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

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.) Other rules might be sound, but these already pose interesting semantic difficulties, and give an interesting account of object equality that suffices for many examples.

3. A Semantics of Objects

In this section and the next one we describe a semantics for our largest calculus, $\mathbf{FOb}_{<:\mu}$, with the associated equational theory. This section concerns an untyped universe and the interpretation of untyped terms. The next section puts a type structure on the untyped universe.

3.1 The Untyped Universe

The technical assumptions on the untyped universe are fairly standard [MacQueen, Plotkin, Sethi 1986; Abadi, Plotkin 1990; Cardone 1990; Amadio 1991]. We need to have a complete partial order $(D,\sqsubseteq)$ such that:

- There are strict, continuous embedding-retraction pairs (e,r) between D and each of $W_\perp$, $(D{\rightarrow}D)$, and $(L{\rightarrow}D)_\perp$,

$$\begin{tikzcd}
W_\perp \arrow[r, "e_w"] & D \arrow[r, "r_w"] & W_\perp \\
(D\rightarrow D) \arrow[r, "e_f"] & D \arrow[r, "r_f"] & (D\rightarrow D) \\
(L\rightarrow D)_\perp \arrow[r, "e_o"] & D \arrow[r, "r_o"] & (L\rightarrow D)_\perp
\end{tikzcd}$$

  where

  – W is a one-point set $\{*\}$; we view $*$ as the error value;
  – $(D{\rightarrow}D)$ is the complete partial order of continuous functions from D to D;
  – L is a countable set of labels $\{m_0, m_1, ...\}$, and $(L{\rightarrow}D)$ 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.

- There is an increasing sequence $p_n: D{\rightarrow}D$ of continuous projections with finite range and with least upper bound the identity. Further, p_0 constantly equals $\perp$. Let ; denote function composition, and $h{\upharpoonright}S$ denote the restriction of h which maps elements outside S to $\perp$. For all i,

$$\begin{array}{rcll}
p_{i+1}(e_w(*)) & = & e_w(*) & \\
p_{i+1}(e_f(f)) & = & e_f(p_i; f; p_i) & f \in D{\rightarrow}D \\
p_{i+1}(e_o(o)) & = & e_o((o; p_i){\upharpoonright}\{m_0,...,m_i\}) & o \in L{\rightarrow}D
\end{array}$$

We commonly view W, $D{\rightarrow}D$, and $L{\rightarrow}D$ as subsets of D, and write $x = *$ for $x = e_w(*)$, $x \in (D{\rightarrow}D)$ for $x \in range(e_f)$, and $x \in (L{\rightarrow}D)$ for $x \in e_o(L{\rightarrow}D)$. When $x, y \in D$, $m \in L$, 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. Thus, $p_{i+1}(*)$ 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 $\perp$ otherwise.

If $x \in D$ 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 $x \sqsubseteq \sqcup_i\, y_i$ 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 $D = W_\perp + (D{\rightarrow}D) + (L{\rightarrow}D)_\perp$, making sure that $p_i(e_o((L{\rightarrow}D)_\perp))$ is always a finite set.

3.2 The Interpretation of Untyped Terms

Next we define the interpretation of untyped terms. The interpretation of typed terms will be based on the interpretation of untyped terms. We define the semantics function:

$$[\![\ ]\!] : (V{\rightarrow}D){\rightarrow}(E{\rightarrow}D)$$

where V is the set of variables and E the set of expressions. We call a mapping ρ in $V{\rightarrow}D$ an environment and write $[\![a]\!]_\rho$ for the semantics of term a with an environment ρ. We write $\langle\langle m_1{=}v_1, ..., m_n{=}v_n\rangle\rangle$ for the function in $(L{\rightarrow}D)$ 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. With this notation, we set:

$$
\begin{array}{rcl}
\llbracket x\rrbracket_\rho & = & \rho(x)\\
\llbracket\lambda(x)b\rrbracket_\rho & = & \lambda(v)\ \llbracket b\rrbracket_{\rho\langle x\leftarrow v\rangle}\\
\llbracket b(a)\rrbracket_\rho & = & \text{if } \llbracket b\rrbracket_\rho ⋿ (D\rightarrow D)\\
 & & \text{then } \llbracket b\rrbracket_\rho(\llbracket a\rrbracket_\rho)\\
 & & \text{else } *\\
\llbracket[m_i{=}\varsigma(x_i)c_i\ ^{i\in I}]\rrbracket_\rho & = & \langle\langle m_i{=}\ \llbracket\lambda(x_i)c_i\rrbracket_\rho\ ^{i\in I}\rangle\rangle\\
\llbracket a.m\rrbracket_\rho & = & \text{if } \llbracket a\rrbracket_\rho ⋿ (L\rightarrow D) \text{ and } \llbracket a\rrbracket_\rho(m) ⋿ (D\rightarrow D)\\
 & & \text{then } \llbracket a\rrbracket_\rho(m)(\llbracket a\rrbracket_\rho)\\
 & & \text{else } *\\
\llbracket a.m \Leftarrow \varsigma(x)c\rrbracket_\rho & = & \text{if } \llbracket a\rrbracket_\rho ⋿ (L\rightarrow D)\\
 & & \text{then } \llbracket a\rrbracket_\rho\langle m\leftarrow\llbracket\lambda(x)c\rrbracket_\rho\rangle\\
 & & \text{else } *
\end{array}
$$

This definition is given in a metalanguage where $v ⋿ V$ is a strict membership test, and where conditionals and conjunctions are strict and evaluated left to right, e.g., $\llbracket a.m \Leftarrow \varsigma(x)c\rrbracket_\rho=\bot$ if $\llbracket a\rrbracket_\rho=\bot$.

Note how the semantics turns ς's into λ's and objects into records. The catch is that the denotations of object types will not be the obvious record types. Note also that, for simplicity, overriding a method does not produce an error if the method is not present in the first place.

This semantics validates the reduction rules for objects. Many of the more interesting equations for objects fail; but they do hold under the typed semantics of the next section.

4. A Semantics of Object Types

The semantics of types is based on the metric approach of MacQueen, Plotkin, and Sethi [MacQueen, Plotkin, Sethi 1986]. More precisely, we follow Amadio [Amadio 1991] 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 $\boldsymbol{F}_{<:\mu}$, and contribute 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, as in $\boldsymbol{F}_{<:\mu}$ [Abadi, Plotkin 1990], it is not clear how to integrate subtyping into the semantics of types as functors.

4.1 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. Section 4.1.1 introduces operations on binary relations. Sections 4.1.2 and 4.1.3 concern the union operation and metric properties.

4.1.1 Semantic Definitions

We will be dealing with binary relations over D, by convention only those that do not have $*$ in their domains. It is easy to show that all our constructions preserve this property. A per is a symmetric transitive binary relation on D. A per X is uniform if u X v implies $(p_i(u))\ X\ (p_i(v))$ for all i. A per 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 cupers.

First we describe some usual operations on cupers. The function-space operation is given by:

$$R\rightarrow T \;=\; \{(f,g) \in (D\rightarrow D)^2 \mid \text{if } xRy \text{ then } f(x)Tg(y)\}$$

If R, T $\in$ **CUPER** then $R\rightarrow T \in$ **CUPER**. We can calculate meets and joins of cupers:

$$\sqcap_{i\in I} X_i \;=\; \cap_{i\in I} X_i \qquad \sqcup_{i\in I} X_i \;=\; \mathcal{C}(\cup_{i\in I} X_i)$$

where $\mathcal{C}(X)$ is the least cuper that contains X. (The cuper $\mathcal{C}(X)$ is always defined.) 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.

In order to give a semantics to object types, we first define:

$$\langle\langle m_i : T_i\ ^{i\in I}\rangle\rangle \;=\; \{(\bot,\bot)\} \cup \{(o,o') \in (L\rightarrow D)^2 \mid \forall i\in I.\ (o(m_i), o'(m_i)) \in T_i\}$$

We view $\langle\langle m_i : T_i\ ^{i\in I}\rangle\rangle$ 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**.

Let **G** denote the set of all cuper functions of the form $\lambda(S)\langle\langle m_i : S\rightarrow T_i\ ^{i\in I}\rangle\rangle$; an element of **G** can be written in this form uniquely. We say that $F = \lambda(S)\langle\langle m_i : S\rightarrow T_i\ ^{i\in I}\rangle\rangle$ extends $H = \lambda(S)\langle\langle m_j : S\rightarrow T_j\ ^{j\in J}\rangle\rangle$ if $I \supseteq J$, and write $F \preccurlyeq H$. We set:

$$(\![m_i{:}T_i\ ^{i\in I}]\!) \;=\; \sqcup\, \{\mu(S)F(S) \mid F \in \mathbf{G},\ F \preccurlyeq \lambda(S)\langle\langle m_i{:}S\rightarrow T_i\ ^{i\in I}\rangle\rangle\}$$

The types of the form $(\![m_i{:}T_i\ ^{i\in I}]\!)$ are our semantic object types. This definition is proper because if $F \in$ **G** then F(S) is contractive in S, and hence $\mu(S)F(S)$ exists and is unique.

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. Here, we define an object type to be a union of recursive record types. Each object type is designed to contain all longer ones. We obtain the expected subtypings: if $I \subseteq J$ then $(\![m_j : T_j\ ^{j\in J}]\!) \subseteq (\![m_i : T_i\ ^{i\in I}]\!)$.

4.1.2 Understanding Unions

In this section we analyze unions of relations. This is necessary because the definitions do not give an explicit description of the elements of a 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. Our first result reduces closure to transitive closure for finite elements. The second one enables us to reason about all elements of a union by reasoning about the elements of its components; this is useful in validating the elimination rules of our calculus for abstract data types and for objects.

Lemma

If $X \in$ **CUPER**, then $\mathcal{C}(X)$ is the chain completion of the transitive closure of the finite part of X.

Lemma

If $R_i \in$ **CUPER** for all $i \in I$, $S \in$ **CUPER**, f and g are continuous functions, and for all i, $(x,y) \in R_i$ implies $(f(x),f(y)) \in S$ and $(f(x),g(y)) \in S$, then $(x,y) \in \sqcup_{i\in I} R_i$ implies $(f(x),g(y)) \in S$.

4.1.3 Metric Properties

Amadio has verified that the usual type constructors $\rightarrow$, $\sqcap$, and μ are contractive or nonexpansive in the cuper model as in the ideal model. In order to interpret all formally contractive type expressions, we extend Amadio's results to deal with bounds, with $\sqcup$, and with object types.

Proposition

If $T_i(R_1,...,R_k)$ is nonexpansive in $R_1,...,R_k$ for all $i \in I$ then $[\![m_i{:}T_i(R_1,...,R_k)^{\,i\in I}]\!]$ is contractive in $R_1,...,R_k$. 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 $\sqcap_{R_{k+1}\in\mathbf{CUPER},\, R_{k+1}\subseteq S(R_1,...,R_k)} T(R_1, ..., R_{k+1})$ and $\sqcup_{R_{k+1}\in\mathbf{CUPER},\, R_{k+1}\subseteq S(R_1,...,R_k)} T(R_1, ..., R_{k+1})$ are contractive (nonexpansive) in $R_1,...,R_k$.

4.2 The Interpretation of Types and Typed Terms

Sections 4.2.1 gives the interpretation of the typed calculus. Section 4.2.2 proves the soundness of the rules under this interpretation. (See the appendix for the syntax of the typed calculus and its rules.)

4.2.1 Interpreting Typed Terms and Types

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:

$$
\begin{aligned}
e(x) &= x\\
e(\lambda(x{:}A)b) &= \lambda(x)e(b)\\
e(b(a)) &= e(b)(e(a))\\
e(\lambda(X{<:}A)b) &= e(b)\\
e(b(A)) &= e(b)\\
e(\text{pack } X{<:}A{=}C,\ b\{X\}{:}B\{X\}) &= e(b)\\
e(\text{open } c \text{ as } X{<:}A,x{:}B \text{ in } d{:}D) &= e(d\{x{\leftarrow}c\})\\
e([m_i{=}\,\varsigma(x_i{:}A)c_i^{\ i\in 1..n}]) &= [m_i{=}\varsigma(x_i)e(c_i)^{\ i\in 1..n}]\\
e(a.m) &= e(a).m\\
e(a.m \Leftarrow \varsigma(x{:}A)\ c) &= e(a).m \Leftarrow \varsigma(x)e(c)
\end{aligned}
$$

We omit the constructs fold and unfold, and value-level recursion. Recursive values can be obtained with the definable **Y** combinator; and since we find exact solutions for recursive-type equations, fold and unfold can both be interpreted as the identity function.

To interpret types, we define the semantics function:

$[\![\ ]\!] : (\mathrm{TV}{\to}\mathbf{CUPER}){\to}(\mathrm{TE}{\to}\mathbf{CUPER})$

where TV is the set of type variables and TE the set of type expressions. A mapping η in $\mathrm{TV}{\to}\mathbf{CUPER}$ is a type environment. We define $[\![A]\!]_\eta$, the semantics of type A with the environment η:

$$
\begin{aligned}
[\![X]\!]_\eta &= \eta(X)\\
[\![A{\to}B]\!]_\eta &= [\![A]\!]_\eta{\to}[\![B]\!]_\eta\\
[\![\forall(X{<:}B)A]\!]_\eta &= \sqcap_{R\in\mathbf{CUPER},\, R\subseteq[\![B]\!]_\eta}\ [\![A]\!]_{\eta\langle X\leftarrow R\rangle}\\
[\![\exists(X{<:}B)A]\!]_\eta &= \sqcup_{R\in\mathbf{CUPER},\, R\subseteq[\![B]\!]_\eta}\ [\![A]\!]_{\eta\langle X\leftarrow R\rangle}\\
[\![\mu(X)A]\!]_\eta &= \mu(T)\ [\![A]\!]_{\eta\langle X\leftarrow T\rangle}\\
[\![[m_i{:}C_i^{\ i\in 1..n}]]\!]_\eta &= [\![m_i : [\![C_i]\!]_\eta^{\ i\in 1..n}]\!]\\
[\![\mathrm{Top}]\!]_\eta &= (D - \{*\})^2
\end{aligned}
$$

Note that the relation <: is simply interpreted as cuper containment. The definition for recursive types is proper because of the connection between contractiveness and formal contractiveness:

Proposition

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.

4.2.2 Soundness of the 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 $(\rho(x),\rho'(x)) \in [\![A]\!]_\eta$. We derive the following soundness results:

Theorem (Soundness)

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	$([\![a]\!]_\rho, [\![a']\!]_{\rho'}) \in [\![A]\!]_\eta$.

An immediate corollary is that no well-typed term has $*$ as its denotation. It also follows that the type theory and the equational theory are consistent.

5. 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 a corresponding extension of System **F**. We adopt their approach, adding existential quantifiers and rows to their target calculus. Our full target calculus appears in the appendix.

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. Such a row can be assembled from the empty row $\emptyset$, which lacks any set of labels, by successive additions of components l:B.

The rows we use are simpler than the ones treated in [Cardelli 1994], since we do not need rows at the value level. 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.

Rows are a convenient but not an essential addition: they can be encoded in $\boldsymbol{F}_{<:}$ [Cardelli 1994], 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.

A derivation Ψ of a judgment $\Im$ is written $\Psi_1,...,\Psi_n \triangleright_R \Im$, where $n\geq 0$, R is the last rule of Ψ, and $\Psi_1,...,\Psi_n$ are the derivations of the assumptions of R; $\overline{\Psi}$ stands for $\Psi_1,...,\Psi_n$. Further, $\overline{\Psi} \triangleright_R \Im$ stands for a derivation of $\Im$ via R, and $\overline{\Psi} \triangleright \Im$ 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
$(\overline{\Psi} \triangleright E \vdash A <: B)^*$	is a term of type $A^*{\to}B^*$ in the target calculus
$(\overline{\Psi} \triangleright E \vdash a : A)^*$	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. Further, it is convenient to rearrange records of functions into functions that return records, so we use types of the form $Y \rightarrow \langle l_i{:}B_i{}^{i\in 1..n},...\rangle$ instead of $\langle l_i{:}Y \rightarrow B_i{}^{i\in 1..n},...\rangle$.

Definition (Translation of Types)

$$\mathit{Top}^* \triangleq \langle\rangle \qquad \text{(the empty record type)}$$
$$[l_i{:}B_i{}^{i\in 1..n}]^* \triangleq \exists(X{::}\uparrow(l_i{}^{i\in 1..n}))\,\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle \qquad X \notin FV(B_i)$$

where $\langle\!\langle l_i{:}B_i{}^{i\in 1..n},X\rangle\!\rangle \triangleq \mu(Y)\ Y \rightarrow \langle l_i{:}B_i{}^{i\in 1..n},X\rangle$, for $Y \notin FV(B_i)\cup\{X\}$.

The translations of subtype and value judgments are by induction on derivations.

Definition (Translation of Subtype Judgments)

$$(\overline{\Psi} \triangleright_{\text{(Sub Refl)}} E \vdash A <: A)^* \triangleq \lambda(x{:}A^*)x$$

$$((\overline{\Psi}_1 \triangleright E \vdash A <: B,\ \overline{\Psi}_2 \triangleright E \vdash B <: C) \triangleright_{\text{(Sub Trans)}} E \vdash A <: C)^* \triangleq$$
$$\quad \lambda(x{:}A^*)\,(\overline{\Psi}_2 \triangleright E \vdash B <: C)^*\,((\overline{\Psi}_1 \triangleright E \vdash A <: B)^*\,(x))$$

$$(\overline{\Psi} \triangleright_{\text{(Sub Top)}} E \vdash A <: \mathit{Top})^* \triangleq \lambda(x{:}A^*)\langle\rangle$$

$$(\overline{\Psi} \triangleright_{\text{(Sub Object)}} E \vdash [l_i{:}B_i{}^{i\in 1..n+m}] <: [l_i{:}B_i{}^{i\in 1..n}])^* \triangleq$$
$$\quad \lambda(o{:}[l_i{:}B_i{}^{i\in 1..n+m}]^*)$$
$$\qquad \text{open } o \text{ as } X{::}\uparrow(l_i{}^{i\in 1..n+m}),\ x{:}\ \langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n+m},X\rangle\!\rangle$$
$$\qquad \text{in pack } X'{::}\uparrow(l_i{}^{i\in 1..n}){=}(l_i{:}B_i{}^{*\,i\in n+1..m},X),\ x{:}\ \langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X'\rangle\!\rangle$$

$$((\overline{\Psi}_1 \triangleright E \vdash a : A,\ \overline{\Psi}_2 \triangleright E \vdash A <: B) \triangleright_{\text{(Val Subsumpiton)}} E \vdash a : B)^* \triangleq$$
$$\quad (\overline{\Psi}_2 \triangleright E \vdash A <: B)^*\,(\overline{\Psi}_1 \triangleright E \vdash a : A)^*$$

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.

Definition (Translation of Value Judgments)

$$(\overline{\Psi} \triangleright_{\text{(Val x)}} E',x{:}A,E'' \vdash x{:}A)^* \triangleq x$$

$$((\overline{\Psi}_i \triangleright E, x_i{:}A \vdash b_i : B_i) \triangleright_{\text{(Val Object)}} E \vdash [l_i{=}\varsigma(x_i{:}A)b_i{}^{i\in 1..n}] : A)^* \triangleq$$
$$\quad \text{pack } X{::}\uparrow(l_i{}^{i\in 1..n}){=}\emptyset,$$
$$\qquad \text{fold}(\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle,$$
$$\qquad\quad \lambda(x{:}\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle)$$
$$\qquad\qquad \langle l_i{=}(\overline{\Psi}_i \triangleright E, x_i{:}A \vdash b_i : B_i)^*$$
$$\qquad\qquad\quad \{x_i \leftarrow \text{pack } X{::}\uparrow(l_i{}^{i\in 1..n}){=}\emptyset,\ x{:}\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle\}$$
$${}^{i\in 1..n}\rangle)$$
$$\qquad : \langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle$$

$$((\overline{\Psi} \triangleright E \vdash a : [l_i{:}B_i{}^{i\in 1..n}]) \triangleright_{\text{(Val Select)}} E \vdash a.l_j : B_j)^* \triangleq$$
$$\quad \text{open } (\overline{\Psi} \triangleright E \vdash a : [l_i{:}B_i{}^{i\in 1..n}])^* \text{ as } X{::}\uparrow(l_i{}^{i\in 1..n}),\ x{:}\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle$$
$$\quad \text{in unfold}(x)(x)\cdot l_j$$

$$((\overline{\Psi}_1 \triangleright E \vdash a : A,\ \overline{\Psi}_2 \triangleright E, x{:}A \vdash b : B_j)$$
$$\triangleright_{\text{(Val Override)}} E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A)^* \triangleq$$
$$\quad \text{open } (\overline{\Psi}_1 \triangleright E \vdash a : [l_i{:}B_i{}^{i\in 1..n}])^* \text{ as } X{::}\uparrow(l_i{}^{i\in 1..n}),\ x{:}\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X\rangle\!\rangle$$
$$\quad \text{in pack } X'{::}\uparrow(l_i{}^{i\in 1..n}){=}X,$$
$$\qquad \text{fold}(\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X'\rangle\!\rangle,$$
$$\qquad\quad \lambda(x'{:}\langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X'\rangle\!\rangle)$$
$$\qquad\qquad \text{unfold}(x)(x')\cdot l_j{:=}$$
$$\qquad\qquad\quad (\overline{\Psi}_2 \triangleright E, x{:}A \vdash b : B_j)^*$$
$$\qquad\qquad\quad\ \{x \leftarrow \text{pack } X''{::}\uparrow(l_i{}^{i\in 1..n}){=}X',\ x'{:}\langle\!\langle l_i{:}B_i{}^{*}$$
$${}^{i\in 1..n},X''\rangle\!\rangle\})$$
$$\qquad : \langle\!\langle l_i{:}B_i{}^{*\,i\in 1..n},X'\rangle\!\rangle$$

We obtain a coherence result, guaranteeing that the translation of a judgment is independent of its derivation.

Theorem (Coherence)

If f and f' are the translations of two $\mathbf{Ob}_{1<:}$ derivations that end with the same value or subtype judgment, then f and f' are provably equal in the target calculus.

Further, the congruence rules and evaluation rules for objects are validated through the translation. So are the equational rules related to subtyping, with the exception of (Eq Sub Object). Proving the translation of (Eq Sub Object) in the target calculus may require additional principles, such as a bisimulation rule for abstract data types [Aczel, Mendler 1989; Plotkin, Abadi 1993].

The translation extends to the full $\mathbf{FOb}_{<:\mu}$. 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].

6. Related work

We finish with some comparisons with the most closely related works.

- 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. 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; Cardelli, Mitchell 1991; Harper, Pierce 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 objects and classes that respects subtyping, but does not account for our method override operations on objects.
- Some ideas presented here originated in the study of Baby Modula-3 [Abadi 1994]. That calculus resembles in power $\mathbf{Ob}_{1<:\mu}$ ($\mathbf{Ob}_{1<:}$ plus recursion), but the two are incomparable. The semantics of Baby Modula-3 is based on ideals. A per semantics is also briefly sketched, without a corresponding study of equational rules.
- Our work is closely related in spirit to that of Mitchell et al. [Mitchell, Honsell, Fisher 1993]. The most significant difference is that they support object extension, while we support subtyping and subsumption. Further, they show soundness of typing by a subject reduction proof, while we construct a model and justify an equational theory.
- The TOOPL language [Bruce 1993] has built-in objects and supports a form of subsumption obtained via two subtyping relations. The semantics of TOOPL is based on generators and F-bounded quantification [Canning, *et al.* 1989], rather than on the self-application semantics. Generator semantics avoid problems of contravariance by binding self at object-formation time with a value-level recursion. As a consequence, objects and object generators (classes) are distinct.

Appendix A: The $\mathbf{Ob}_{1<:}$ Calculus

Environments	$E ::= \emptyset \mid E,x{:}A$
Type Constants	K
Types	$A,B,C ::= K \mid Top \mid [l_i{:}B_i{}^{i\in 1..n}]$
Variables	x,y
Constants	k

Values $a,b,c ::= x \mid k(a_i{}^{i\in 1..n})$
$\mid [l_i{=}\varsigma(x_i{:}A)b_i{}^{i\in 1..n}] \mid a.l \mid a.l{\Leftarrow}\varsigma(x{:}A)b$

In addition to the rules for constants [Abadi, Cardelli 1994b] and the rules give in the text, (Type Top), (Type Object), (Sub Top), (Sub Object), (Val Subsumption), (Val x), (Val Object), (Val Select), (Val Override), (Eq Subsumption), (Eq Top), (Eq Sub Object), (Eval Select), and (Eval Override), we have:

(Env ∅)
$$\frac{}{\emptyset \vdash \diamond}$$

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

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

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

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

Appendix B: The $\mathbf{FOb}_{<:\mu}$ Calculus

Environments $E ::= \emptyset \mid E,x{:}A \mid E,X{<:}A$
Type Variables X,Y
Types $A,B,C,D ::= X \mid Top \mid [l_i{:}B_i{}^{i\in 1..n}] \mid A{\to}B$
$\mid \forall(X{<:}A)B \mid \exists(X{<:}A)B \mid \mu(X)A$
Variables x,y
Values $a,b,c,d ::= x \mid [l_i{=}\varsigma(x_i{:}A)b_i{}^{i\in 1..n}] \mid a.l \mid a.l{\Leftarrow}\varsigma(x{:}A)b$
$\mid \lambda(x{:}A)b \mid b(a) \mid \lambda(X{<:}A)b \mid b(A)$
$\mid$ pack $X{<:}A{=}C, b\{X\}{:}B\{X\}$
$\mid$ open c as $X{<:}A,x{:}B$ in $d{:}D$
$\mid fold(\mu(X)A, a) \mid unfold(a) \mid \mu(x{:}A)a$

This calculus consists of the rules of $\mathbf{Ob}_{1<:}$, except for the rules for constants, plus the following:

(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 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<:)
$$\frac{E,X \vdash A \quad A{\succ}X}{E \vdash \mu(X)A}$$

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

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

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

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

(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)

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

(Eval Eta)

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

(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<:)

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

(Eval Unpack<:) (where c≡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)

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

Formal Contractiveness.

The relation $A \succ Y$ (type expression A is formally contractive in variable Y) is defined as follows:

$X \succ Y$	if $X\neq Y$	
$Top \succ Y$	always	
$[l_i:B_i\ ^{i\in 1..n}] \succ Y$	always	
$A\to B \succ Y$	always	
$\forall(X<:A)B \succ Y$	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)
$\mu(X)A \succ Y$	if $A \succ Y$	

Appendix C: 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 Vars	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	$a,b,c,d ::= x \mid \langle l_i=b_i\ ^{i\in 1..n}\rangle \mid a\cdot l \mid a\cdot l:=b \mid \lambda(x:A)b \mid b(a) \mid \lambda(X::K)b \mid b(A) \mid \text{pack } X::K=C, b\{X\}:B\{X\} \mid \text{open } c \text{ as } X::K,x:B \text{ in } d:D \mid \text{fold}(\mu(X)A, a) \mid \text{unfold}(a) \mid \mu(x:A)a$

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; we write $(l,l_1..l_n)$ to imply that $l \neq l_1..l_n$.

(Env ∅)

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

(Env x)

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

(Env X)

$$\frac{E \vdash K \text{ kind} \quad X\notin dom(E)}{E,X::K \vdash \diamond}$$

(Kind Type)

$$\frac{E \vdash \diamond}{E \vdash Type \text{ kind}}$$

(Kind Row)

$$\frac{E \vdash \diamond}{E \vdash \uparrow(l_1..l_n) \text{ kind}}$$

(Type X)

$$\frac{E',X,E'' \vdash \diamond}{E',X,E'' \vdash X}$$

(Type Arrow)

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

(Type Record)

$$\frac{E \vdash R::\uparrow()}{E \vdash \langle R\rangle}$$

(Type All)

$$\frac{E,X::K \vdash B}{E \vdash \forall(X::K)B}$$

(Type Exists)

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

(Type Rec)

$$\frac{E,X \vdash A \quad A\succ X}{E \vdash \mu(X)A}$$

(Row X)

$$\frac{E',X::\uparrow(l_1..l_n),E'' \vdash \diamond}{E',X::\uparrow(l_1..l_n),E'' \vdash X::\uparrow(l_1..l_n)}$$

(Row ∅)

$$\frac{E \vdash \diamond}{E \vdash \emptyset :: \uparrow(l_1..l_n)}$$

(Row Cons)

$$\frac{E \vdash R::\uparrow(l,l_1..l_n) \quad E \vdash B}{E \vdash l:B,R :: \uparrow(l_1..l_n)}$$

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

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

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

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

(Eq Record)
$$\frac{E \vdash b_i \leftrightarrow b_i' : B_i \qquad \forall i \in 1..n}{E \vdash \langle l_i = b_i{}^{i \in 1..n}\rangle \leftrightarrow \langle l_i = b_i'{}^{i \in 1..n}\rangle : \langle l_i{:}B_i{}^{i \in 1..n}, \emptyset\rangle}$$

(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)
$$\frac{E \vdash a \leftrightarrow a' : \langle l{:}B, R\rangle \qquad E \vdash b \leftrightarrow b' : B}{E \vdash a \cdot l := b \leftrightarrow a' \cdot l := b' : \langle l{:}B, R\rangle}$$

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

(Eq Pack)
$$\frac{E \vdash A{::}K \qquad E, X{::}K \vdash B\{X\} \qquad E \vdash b\{A\} \leftrightarrow b'\{A\} : B\{A\}}{E \vdash (\mathrm{pack}\ X{::}K{=}A, b\{X\}{:}B\{X\}) \leftrightarrow (\mathrm{pack}\ X{::}K{=}A, b'\{X\}{:}B\{X\}) : \exists(X{::}K)B\{X\}}$$

(Eq Open)
$$\frac{E \vdash c \leftrightarrow c' : \exists(X{::}K)B \qquad E \vdash D \qquad E, X{::}K, x{:}B \vdash d \leftrightarrow d' : D}{E \vdash (\mathrm{open}\ c\ \mathrm{as}\ X{::}K, x{:}B\ \mathrm{in}\ d{:}D) \leftrightarrow (\mathrm{open}\ c'\ \mathrm{as}\ X{::}K, x{:}B\ \mathrm{in}\ d'{:}D) : D}$$

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

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

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

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

(Eval Select) (where $a \equiv \langle l{=}b, l_i{=}b_i{}^{i \in 1..n}\rangle$)
$$\frac{E \vdash a : \langle l{:}B, l_i{:}B_i{}^{i \in 1..n}, \emptyset\rangle}{E \vdash a \cdot l \leftrightarrow b : B}$$

(Eval Override) (where $a \equiv \langle l{=}b, l_i{=}b_i{}^{i \in 1..n}\rangle$)
$$\frac{E \vdash a : \langle l{:}B, l_i{:}B_i{}^{i \in 1..n}, \emptyset\rangle \qquad E \vdash b' : B}{E \vdash a \cdot l := b' \leftrightarrow \langle l{=}b', l_i{=}b_i{}^{i \in 1..n}\rangle : \langle l{:}B, l_i{:}B_i{}^{i \in 1..n}, \emptyset\rangle}$$

(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)
$$\frac{E \vdash \lambda(X{::}K)b : \forall(X{::}K)B \qquad E \vdash A{::}K}{E \vdash (\lambda(X{::}K)b)(A) \leftrightarrow b\{X \leftarrow A\} : B\{X \leftarrow A\}}$$

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

(Eval Unpack) (where $c \equiv \mathrm{pack}\ X{::}K{=}A, b\{X\}{:}B\{X\}$)
$$\frac{E \vdash c : \exists(X{::}K)B\{X\} \qquad E \vdash D \qquad E, X{::}K, x{:}B\{X\} \vdash d\{X,x\} : D}{E \vdash (\mathrm{open}\ c\ \mathrm{as}\ X{::}K, x{:}B\{X\}\ \mathrm{in}\ d\{X,x\}{:}D) \leftrightarrow d\{A, b\{A\}\} : D}$$

(Eval Repack)
$$\frac{E \vdash b : \exists(X{::}K)B\{X\} \qquad E, y{:}\exists(X{::}K)B\{X\} \vdash d\{y\} : D}{E \vdash (\mathrm{open}\ b\ \mathrm{as}\ X{::}K, x{:}B\{X\}\ \mathrm{in}\ d\{\mathrm{pack}\ X'{::}K{=}X, x{:}B\{X'\}\}{:}D) \leftrightarrow d\{b\} : D}$$

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

(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
$A \to B \succ Y$	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, Cardelli 1994a] M. Abadi and L. Cardelli, **A theory of primitive objects**. *To appear*.

[Abadi, Cardelli 1994b] M. Abadi and L. Cardelli. **A theory of primitive objects: second-order systems**. *Proc. ESOP'94 - European Symposium on Programming*. Springer-Verlag.

[Abadi, Cardelli 1994c] M. Abadi and L. Cardelli. **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag.

[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 1991] R.M. Amadio, **Recursion over realizability structures**. *Information and Computation* **91**(1), 55-85.

[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.

[Canning, *et al.* 1989] P. Canning, W. Cook, W. Hill, W. Olthoff, and J.C. Mitchell. **F-bounded polymorphism for object-oriented programming**. *Proc. ACM Conference on Functional Programming and Computer Architecture*.

[Cardelli 1994] L. Cardelli, **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming*, C.A. Gunter and J.C. Mitchell, ed. MIT Press (to appear). Also Technical Report n.81, Digital Systems Research Center, 1991.

[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.

[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.

[Girard, Lafont, Taylor 1989] J.-Y. Girard, Y. Lafont, and P. Taylor, **Proofs and types**. Cambridge University Press.

[Harper, Pierce 1991] R. Harper and B. Pierce. **A record calculus based on symmetric concatenation**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*.

[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.

[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*.

[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 1993] G.D. Plotkin and M. Abadi. **A logic for parametric polymorphism**. *Proc. International Conference on Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science 664. Springer-Verlag.

[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.

[Wand 1989] M. Wand. **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*.