# Multiple Inheritance via Intersection Types

**Adriana B. Compagnoni**
**Catholic University, Nijmegen**

**Benjamin C. Pierce**
**LFCS, Edinburgh**

**August 30, 1993**

**University of Edinburgh**
**Technical Report ECS-LFCS-93-275**
**and**
**Catholic University Nijmegen**
**C.S. Department Technical Report 93-18**

**Abstract**

Combining intersection types with higher-order subtyping yields a typed model of object-oriented programming with multiple inheritance. Objects, message passing, subtyping, and inheritance appear as programming idioms in a typed $\lambda$-calculus, a modelling technique that facilitates experimentation and helps in distinguishing between essential aspects of the object-oriented style —encapsulation and subtype polymorphism, which are directly reflected in the low-level type system — and useful but inessential programming idioms such as inheritance.

The target calculus, a natural generalization of system $F^\omega$ with intersection types, is of independent interest. We establish basic structural properties and give a proof of type soundness using a simple semantics based on partial equivalence relations.

## 1 Introduction

Among the first applications of subtyping in $\lambda$-calculi was modeling the refinement of interfaces in object-oriented languages [Car88]. The formal subtype relation $\Gamma \vdash S \leq T$ models the assertion that the objects in some collection $S$ provide more services than those in $T$, so that it is safe to use a member of $S$ in any context where a member of $T$ is expected.

Subtyping itself is a straightforward concept, but it took years to build full-scale models of existing object-oriented languages on this foundation. Treating the interaction between interface refinement and encapsulation of objects has required higher-order generalizations of subtyping — the F-bounded quantification of Canning, Cook, Hill, Olthoff, and Mitchell [CCH+89, CHC90] or Cardelli and Mitchell's system $F^{\omega}_{\leq}$ [Car90, Mit90a, BM92]. Using these techniques, a number of fairly complete models of languages such as Smalltalk [GR83] and Self [US87] have recently appeared [Bru93, MHF93, PT93]. The current explosion of work in this area also includes models of the object-oriented features of languages such as Modula-3 [Aba93] and CLOS [CGL92, Cas92], as well as some foundational studies [Car92, AC93, HP92], which are less closely related to the results described here.

The common goal of these studies is to prove the safety of a type system describing a set of high-level syntactic constructs for object encapsulation, message passing, and inheritance. Most proceed by mapping the high-level syntax directly into a denotational model and proving the soundness of this interpretation using standard semantic techniques [Bru93, MHF93, Aba93, AC93]. Others translate the high-level syntax into a more conventional $\lambda$-calculus, whose own type-safety is established separately; the soundness of the typing rules for the object features then follows from the soundness of the target system [CGL92, Cas92, Car92, PT93]. Both approaches have advantages, but the latter can often be simpler to use and understand, since it allows the implementation of complex mechanisms like inheritance to be considered as *programming* in a typed language as opposed to definition of a complex semantic interpretation function. We can verify type safety automatically using a typechecker for the underlying $\lambda$-calculus, and we can experiment easily with alternatives [Pie93a, Pie93b].

The object model of Pierce and Turner [PT93], on which the present study is based, encodes objects as expressions of $F^{\omega}_{\leq}$, an extension of Girard's System $F^{\omega}$ [Gir72] with bounded quantification. Given a description `M` of a public interface — the names and types of a set of methods — the type `Object(M)` denotes the type of objects satisfying this description.[1] For example, if `PointM` describes the interface of one-dimensional point objects responding to the messages `setX` and `getX` then `Object(PointM)` is the type of points. Associated with each such collection of objects is a group of functions for sending messages, with types like:

```
Point'setX  :  All(M < PointM)
                    Object(M) -> Int -> Object(M)
```

The bounded quantifier `All(M<PointM)` expresses the fact that the message `setX` can actually be sent to any object whose interface refines the interface of points. Given such an object and an integer representing its new x-coordinate, `Point'setX` returns a new object with an appropriately updated position. (Note that we are working here with a functional style of object-oriented programming, where methods like `setX` return new objects instead of modifying existing ones. The prospects for extending our model to objects with mutable state are discussed in Section 6.)

The foregoing accounts for the fundamental features of object encapsulation and interface refinement (and correctly handles their interaction; this is the difficult part). But it omits some characteristic features of popular object-oriented languages, notably inheritance.

In general terms, inheritance is a mechanism allowing the implementations of different sorts of objects that share some of their behavior to be factored so that the common behavior is written just once. Of the three common varieties of inheritance — subclassing as in Smalltalk, delegation as in Self, and generic functions as in CLOS — subclassing and delegation

[1] Technically, object interfaces are type operators of kind $\star\to\star$, maps from types to types, and `Object` $\in (\star\to\star) \to \star$ is a higher-order constructor. See Sections 2 and 5 for details and [PT93] for a longer discussion.

can be handled by similar techniques, while generic functions seem to require substantially different foundations. We consider a class-based language here.

A class is a data structure that can be used in two ways: as a template for creating new objects, and as the basis for defining subclasses by incremental extension. If `M` is an object interface, then `Class(M)` is the type of classes whose instances are objects of type `Object(M)`. The polymorphic function `new` maps a class into a new object:

```
new : All(M:*->*) Class(M) -> Object(M)
```

Another function, `extend`, takes an existing class and a description of some new methods and constructs a new class combining the old and new behaviors:

```
extend : All(SuperM:*->*)
         All(SelfM<SuperM)
           Class(SuperM) -> ... -> Class(SelfM)
```

(The details hidden here by ... are revealed in the following section and Section 5.) The bounded quantifier `All(SelfM<SuperM)` ensures that the interface of the new class refines that of the old: `Object(SelfM)` is a subtype of `Object(SuperM)`.[2]

To handle multiple inheritance in this setting, we must enrich the `extend` function to take two or more superclasses as arguments. Consider the case where the new class inherits methods from two superclasses. This version — call it `extend2` (there will be an analogous one for each $n$) — should have a type like:

```
extend2 : All(SuperM1:*->*)       % the interface of one superclass
          All(SuperM2:*->*)       % the interface of the other superclass
          All(SelfM<???)          % the interface of the class being built
               Class(SuperM1)     % the first superclass
            -> Class(SuperM2)     % the second superclass
            -> ...                % (how to build the new class)
            -> Class(SelfM)       % the new class itself
```

But the upper bound of `SelfM` presents a problem: we must ensure that `SelfM` is a subtype of *both* `SuperM1` and `SuperM2`, which falls outside the expressive scope of our target $\lambda$-calculus $F^{\omega}_{\leq}$.

[2] There is a delicate technical point here. Cook, Canning, and Hill have argued [CHC90] that "subtyping is not inheritance," that the two concepts live in different categories — subtyping in the world of types, inheritance in the world of values — and must be considered separately to avoid confusion. It might appear, therefore, that we should be able to use inheritance to form new classes with arbitrary interfaces, not only subclasses with more refined interfaces. Indeed, this would be true if it were not for the rather subtle form of self-reference inherent in "sending messages to `self`" in most object-oriented languages, which we are also modeling here. A method defined in a superclass may refer, via `self`, to a different method of the same object. If this method is overriden in the definition of a subclass, the new behavior will be invoked instead. This overriding can only be type-safe if the the new method has a more refined type than the old one. Thus, in the presence of `self`, the inheritance hierarchy must follow the subtype hierarchy.

Intuitively, what we want to write is

```
extend2 : All(SuperM1:*->*) All(SuperM2:*->*)
          All(SelfM < SuperM1 "and" SuperM2)
              Class(SuperM1)
            -> Class(SuperM2)
            -> ...
            -> Class(SelfM)
```

where, informally, "and" forms the conjunction of the two superclass specifications. Fortunately, a type constructor with exactly this meaning has already appeared in the literature. First-order type systems with intersection types have been investigated by the group in Torino [CDC78, BCD83] and elsewhere. (See [CC90] for background and further references.) A second-order $\lambda$-calculus with intersection types was studied by Pierce [Pie91]. The calculus needed here is the $\omega$-order extension of this system.

A type system combining intersection types with a powerful form of polymorphism is of independent interest. Reynolds [Rey88] has argued that intersection types can form the basis of elegant language designs. But his Forsythe language has only a first-order type system, and thus lacks some of the expressive possibilities of polymorphic languages like ML. Our work represents a step toward a synthesis of these styles of language design.

The following section shows some examples of multiple inheritance using a simple high-level syntax. Section 3 defines the $\lambda$-calculus $F^{\omega}_{\wedge}$, an extension of System $F^{\omega}$ with intersection types, and establishes some of its basic properties. In Section 4 we prove the paper's main theoretical result: the soundness of typing and subtyping for a model based on partial equivalence relations. Section 5 develops an implementation of inheritance in this setting. Section 6 offers further comparisons with related work and suggestions for future research.

# 2 An Example of Multiple Inheritance

We begin by recalling the encodings of some basic concepts of object-oriented programming in $F^{\omega}_{\leq}$ and showing a simple example of multiple inheritance in this setting.

In this setting, an object *interface specification* is modeled as a function from types to types, describing the behaviors of a collection of methods as transformations on the object's internal state. For example, the interface of one-dimensional point objects supporting the messages `getX`, `setX`, and `bump` is captured by the type operator

```
# PointM = Fun(Rep) {| setX: Rep->Int->Rep, getX: Rep->Int, bump: Rep->Rep |};
PointM : *->*
```

which expresses the fact that the `getX` method of a point interrogates its internal state and returns an integer, that the `set` method transforms the internal state and a new position into an updated internal state, and that `bump`, which increases the position by one, maps one internal state to another. (The `#` in the left-hand margin indicates that this expression has been checked by our implementation; the typechecker's response follows.) The abstraction over the type `Rep` of the internal state hides the actual internal state from outside view. Concretely, a point whose internal state type is `{|x:Int|}` — a one-field record containing an integer — will contain a record of methods with types

```
{|  setX: {|x:Int|} -> Int -> {|x:Int|},
    getX: {|x:Int|} -> Int,
    bump: {|x:Int|} -> {|x:Int|}         |}
```

while a point whose internal state type is richer, say `{|x:Int,y:Int|}`, will have correspondingly richer concrete types for its methods:

```
{|  setX: {|x:Int,y:Int|} -> Int -> {|x:Int,y:Int|},
    getX: {|x:Int,y:Int|} -> Int,
    bump: {|x:Int,y:Int|} -> {|x:Int,y:Int|}         |}
```

Externally, we expect the difference between these two to be invisible; thus, the public interface to the methods, `PointM`, abstracts away from any particular representation type. Both point objects are elements of the type `Object(PointM)`. (For present purposes, it is not important how `Object` itself is defined. C.f. [PT93, HP92].)

New objects are created by applying the polymorphic function `new` to a *class*. Given an interface `M` and a class for this interface — that is, a class whose *instances* are objects with interface `M` — `new` creates and returns such an object. New classes, in turn, are created by applying the polymorphic function `extend` to an existing class along with a specification of an incremental change to its behavior:

```
extend = <val>
       : All(SuperM)
         All(SelfM<SuperM)
         All(SelfDiffR)
              (Class SuperM)
           -> SelfDiffR
           -> (All(FinalR)
                   (Extractor FinalR SelfDiffR)
                 ->(SuperM FinalR)
                 ->(SelfM FinalR)
                 ->(SelfM FinalR))
           -> (Class SelfM)
```

In detail, the arguments expected by `extend` comprise:

- The interface `SuperM` of the existing class.
- The interface `SelfM` of the new class that will be returned by `extend`.
- The type `SelfDiffR`, which describes the difference between the representation of the superclass (whatever it may be) and the representation of the new class. In conventional terminology, this is the set of new instance variables introduced by the subclass.
- The superclass itself — an element of `Class(SuperM)` (our typechecker prints it as `Class SuperM`).
- An initial value — an element of `SelfDiffR` — for the new part of the state.

- A polymorphic "method builder" function.

Given all these, `extend` returns a class for the interface `SelfM`.

The method builder function, which does the work of constructing the vector of methods to be used in instances of the new class, must itself take several parameters:

- The "final" representation type `FinalR`, which is fixed at the moment when `new` is applied to a class.

- An "extractor," which provides a mapping back and forth between the final representation type and the local representation type, allowing the local methods to access the part of the state that interests them.

- The "super methods" of the existing class.

- The "self methods" of the new class, which are used to model the characteristic object-oriented feature of "sending a message to `self`."

Given these, the method builder must return a collection of methods for the new object.

For uniformity, let us assume that there is just one base class — the class of "things," whose instances are objects with no behavior at all:

```
# ThingM = Fun(Rep) {| |};
ThingM : *->*
thingClass = <val> : Class ThingM
```

To build a class of points extending `thingClass`, we first choose the "local" part of the representation of points.

```
# PointDiffR = {| x:Int |};
PointDiffR : *
```

Now we create `pointClass` by applying `extend` as follows. (See Section 5 for more details.)

```
# pointClass =
#   extend
#     ThingM                              % superclass interface
#     PointM                              % interface for new class
#     PointDiffR                          % local state type
#     thingClass                          % the superclass itself
#     { x = 0 }                           % initial value for local state
#     (fun(FinalR)                        % "method builder" function...
#      fun(e: Extractor FinalR PointDiffR)     % mapping a "state extractor"
#      fun(super: ThingM FinalR)               % and the "super methods"
#      fun(self: PointM FinalR)                % and the "self methods"
#        {getX = fun(s:FinalR)                 % to a getX method
#                  (e.get s).x,                %   that returns the local x field
#         setX = fun(s:FinalR)                 % and a setX method
#                  fun(i:Int) e.put s {x=i},   %   that overwrites the x field
```

```
#         bump = fun(s:FinalR)                     % and a bump method
#                  self.setX s                     %   that calls setX on self to set x
#                    (plus (self.getX s) 1)        %   to one more than self.getX
#        });
pointClass = <val> : Class PointM
```

Of course, this definition of pointClass is quite verbose. It is not hard to design higher-level syntax for objects, message passing, and class extension that looks like ordinary object-oriented source code, but since we are building a foundational model here, we prefer the low-level notation.

Similarly, we can define the interface for "colored objects" — objects supporting the messages setC and getC — as follows:

```
# ColoredM = Fun(Rep) {| setC: Rep->Color->Rep, getC: Rep->Color |};
ColoredM : *->*
```

Again, one instance variable suffices to represent the color of a colored object:

```
# ColoredDiffR = {| c:Color |};
ColoredDiffR : *
```

A class of colored objects can now be created by extending thingClass as we did to build pointClass:

```
# coloredClass =
#   extend ThingM ColoredM ColoredDiffR thingClass
#   { c = black }
#   (fun(FinalR)
#    fun(e: Extractor FinalR ColoredDiffR)
#    fun(super: ThingM FinalR)
#    fun(self: ColoredM FinalR)
#      {getC = fun(s:FinalR) (e.get s).c,
#       setC = fun(s:FinalR) fun(newc:Color) e.put s {c=newc}
#      });
coloredClass = <val> : Class ColoredM
```

Now we have reached the point where we can use *multiple* inheritance to combine the classes of point objects and colored objects, yielding a new class of colored points. The interface of colored points contains all the messages of both superclasses:

```
# CPointM = Fun(Rep) {| setX: Rep->Int->Rep, getX: Rep->Int, bump: Rep->Rep,
#                       setC: Rep->Color->Rep, getC: Rep->Color |};
CPointM : *->*
```

For this simple implementation, no additional instance variables are needed: we can set `CPointDiffR = {| |}`.

To make the example more interesting, we take the methods getX, setC, and getC unchanged from the superclasses, while *overriding* the definition of setX so that, in addition to setting the x coordinate as usual, it also sets the color to, say, blue:

```
# cpointClass =
#   extend2 PointM ColoredM CPointM
#     CPointDiffR pointClass coloredClass { }
#     (fun(FinalR)
#      fun(e: Extractor FinalR CPointDiffR)
#      fun(super1: PointM FinalR)
#      fun(super2: ColoredM FinalR)
#      fun(self: CPointM FinalR)
#        {setX = fun(s:FinalR) fun(i:Int) % the new setX method:
#          let s1 = super1.setX s i in    % use pointClass's setX to set position
#          let s2 = super2.setC s1 blue   % and coloredClass's setC to set color
#          in s2   end end,
#         getX = super1.getX,             % copy all the remaining methods
#         bump = super1.bump,             % from the appropriate superclass
#         setC = super2.setC,
#         getC = super2.getC
#        });
cpointClass = <val> : Class CPointM
```

Here, the low level at which we are working is reflected in the fact that the old methods `getX`, `bump`, `setC`, and `getC` must be copied explicitly from the superclasses to the new class. Introducing high-level syntax for multiple inheritance would, of course, raise all the usual questions (must each inherited method appear in only one of the superclasses? if it appears in more than one, which should be copied to the subclass? etc.), for which the usual solutions will apply.

To test what we have done, let's build a colored point and send it some messages:

```
# p = new CPointM cpointClass;
p = <val> : Object CPointM
# Colored'getC CPointM p;
black : Color
# p1 = Point'bump CPointM p;
p1 = <val> : Object CPointM
# Point'getX CPointM p1;
1 : Int
# Colored'getC CPointM p1;
blue : Color
```

Note that sending our colored point the `bump` method has the effect of changing its color to blue: the overriden behavior of the `setX` method is observable in the behavior of `bump` method, even though `bump` was not redefined in the subclass.

# 3 Syntax

We will return to the implementations of `Class` and `extend2` in Section 5. But first, let us make a more formal presentation of the typed $\lambda$-calculus in which we are working.

## 3.1 Notational Preliminaries

The *kinds* of $F^\omega_\wedge$ are those of $F^\omega$: the kind $\star$ of proper types and the kinds $K_1 \to K_2$ of functions on types (sometimes called type operators).

$$\begin{array}{lll} K & ::= & \star & \text{types} \\ & & K \to K & \text{type operators} \end{array}$$

The language of *types* of $F^\omega_\wedge$ is a straightforward higher-order extension of $F_\leq$, Cardelli and Wegner's second-order calculus of bounded quantification. Like $F_\leq$, it includes type variables (written $X$), function types ($T \to T'$), and polymorphic types ($\text{All}\,(X{\leq}T{:}K)\,T'$), in which the bound type variable $X$ ranges over all subtypes of the upper bound $T$. Moreover, like $F^\omega$, we allow types to be abstracted on types ($\text{Fun}\,(X{:}K)\,T$) and applied to argument types ($T\,T'$); in effect, these forms introduce a simply typed $\lambda$-calculus at the level of types. Finally, we allow arbitrary finite intersections ($\bigwedge^K [T_1..T_n]$), where all the $T_i$'s are members of the same kind $K$.

$$\begin{array}{llll} T & ::= & X & \text{type variable} \\ & & T \to T & \text{function type} \\ & & \text{All}\,(X{\leq}T{:}K)\,T & \text{polymorphic type} \\ & & \text{Fun}\,(X{:}K)\,T & \text{operator abstraction} \\ & & T\,T & \text{operator application} \\ & & \bigwedge^K [T..T] & \text{intersection at kind } K \end{array}$$

We use the abbreviations $\top^K$ and $S \wedge^K T$ for nullary and binary intersection:

$$\begin{array}{lll} \top^K & \triangleq & \bigwedge^K [\,] \\ S \wedge^K T & \triangleq & \bigwedge^K [S, T] \end{array}$$

We drop the maximal type *Top* of $F_\leq$, since its role is played here by the empty intersection $\top^\star$.

For technical convenience, we provide kind annotations on bound variables and intersections so that every type has an "obvious kind," which can be read off directly from its structure and the kind declarations in the context. In examples, however, we often omit these kind annotations. Our implementation includes a simple form of "kind inference" to support this convention. Also, we omit existential types and records from our formal treatment; existentials can be encoded in terms of universal quantifiers, and records can be treated using standard techniques [Rey88].

The language of terms includes the variables ($x$), functional abstractions ($\text{fun}\,(x{:}T)\,e$), and applications ($e\,e$) of the simply typed $\lambda$-calculus, plus the type abstraction ($\text{fun}\,(X{\leq}T{:}K)\,e$) and application ($e\,T$) of $F^\omega$, where, as in $F_\leq$, each type variable is given an upper bound at the point where it is introduced.

$$\begin{array}{llll} e & ::= & x & \text{variable} \\ & & \text{fun}\,(x{:}T)\,e & \text{abstraction} \\ & & e\,e & \text{application} \\ & & \text{fun}\,(X{\leq}T{:}K)\,e & \text{type abstraction} \\ & & e\,T & \text{type application} \\ & & \text{for}\,(X \in T..T)\,e & \text{alternation} \end{array}$$

Intersection types are introduced by expressions of the form "for $(X \in T_1..T_n)\, e$," which can be read as instructions to the typechecker to analyze the expression $e$ separately under the assumptions $X \equiv T_1$, $X \equiv T_2$, ..., etc. and conjoin the results. For example, if Int$\leq$Real and $+ \in$ Int$\to$Int$\to$Int $\wedge$ Real$\to$Real$\to$Real, then we can derive

$$\text{for}\,(X \in \text{Int, Real})\ \ \text{fun}\,(x{:}X)\ \ x + x \quad \in \quad \text{Int}\to\text{Int} \wedge \text{Real}\to\text{Real}.$$

(The "for" construct is not actually needed in our implementation of multiple inheritance, since we only use intersections to constrain one type to be a subtype of two others simultaneously. We include it here because we are also interested in $F^{\omega}_{\wedge}$ in its own right.)

A *context* $\Gamma$ is a finite sequence of typing and/or subtyping assumptions for a set of variables and/or type variables. The empty context is written $\emptyset$.

| | |
|---|---|
| $\emptyset$ | empty context |
| $\Gamma,\ x{:}T$ | term variable declaration |
| $\Gamma,\ X{\leq}T{:}K$ | type variable declaration |

When writing nonempty contexts, we omit the initial $\emptyset$. The domain of $\Gamma$ is written dom($\Gamma$). The functions FV(—) and FTV(—) give the sets of free variables and free type variables of a term, type, or context. Since we are careful to ensure that no variable is bound more than once, we sometimes abuse notation and consider contexts as finite functions: $\Gamma(X)$ yields the bound of $X$ in $\Gamma$, where $X$ is implicitly asserted to be in dom($\Gamma$).

Types, terms, contexts, statements, and derivations that differ only in the names of bound variables are considered identical. (The underlying idea is that variables are de Bruijn indexes [dB72]).

The capture avoiding substitution of $S$ for $X$ in $T$ is written $T[X{\leftarrow}S]$. Substitution is written similarly for terms, and is extended pointwise to contexts.

The usual one-step $\beta$-reduction of a type $S$ to a type $T$ is written $\to_\beta$. Conversion is written $=_\beta$.

## 3.2 Typing Rules

We now present the rules for kinding, subtyping, and typing in $F^{\omega}_{\wedge}$. They are organized as proof systems for four interdependent judgement forms:

| | |
|---|---|
| $\Gamma \vdash \text{ok}$ | well-formed context |
| $\Gamma \vdash T \in K$ | well-kinded type |
| $\Gamma \vdash S \leq T$ | subtype |
| $\Gamma \vdash e \in T$ | well-typed term. |

We sometimes use the metavariable $\Sigma$ to range over statements (right-hand sides of judgements) of any of these four forms.

Most of the rules include premises of two rather different sorts: *structural premises*, which play an essential role in giving the rule its intended semantic force, and *well-formation premises*, which ensure that the entities named in the rule are of the expected sorts. In an algorithmic presentation of the system (on which an implementation might be based), the well-formation premises would be replaced by the meta-theoretic observation that "recursive calls" in the premises of all the rules preserve the well-formedness of the "arguments" named in the conclusion.

In the interest of brevity, we omit well-formation premises that can be derived from others. For example, in the rule S-ARROW, we drop the premise $\Gamma \vdash T_1 \rightarrow T_2 \in \star$, since it follows from $\Gamma \vdash S_1 \rightarrow S_2 \in \star$ using the properties proved in Section 3.3.

### 3.2.1 Contexts

The rules for well-formed contexts are the usual ones: a start rule for the empty context and rules allowing a given well-formed context to be extended with either a term variable binding or a type variable binding.

$$\emptyset \vdash \mathrm{ok} \tag{C-Empty}$$

$$\frac{\Gamma \vdash T \in \star \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma,\ x{:}T \vdash \mathrm{ok}} \tag{C-Var}$$

$$\frac{\Gamma \vdash T \in K \qquad X \notin \mathrm{dom}(\Gamma)}{\Gamma,\ X{\leq}T{:}K \vdash \mathrm{ok}} \tag{C-TVar}$$

### 3.2.2 Type Formation

For each type constructor, we give a rule specifying how it can be used to build well-formed type expressions. The critical rules are K-OABS and K-OAPP, which form type abstractions and type applications (essentially as in a simply typed $\lambda$-calculus).

Note that we extend the context with $\top^{K_1}$ in K-ALL instead of $T_1$, underscoring the fact that "non-administrative" changes to the context are made only by the typing rules. (C.f. S-ALL below.)

The well-formedness premise $\Gamma \vdash \mathrm{ok}$ in K-MEET (and in T-MEET below) is required for the case where $n = 0$.

$$\frac{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \mathrm{ok}}{\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash X \in K} \tag{K-TVar}$$

$$\frac{\Gamma \vdash T_1 \in \star \qquad \Gamma \vdash T_2 \in \star}{\Gamma \vdash T_1 \rightarrow T_2 \in \star} \tag{K-Arrow}$$

$$\frac{\Gamma \vdash T_1 \in K_1 \qquad \Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star}{\Gamma \vdash \mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2 \in \star} \tag{K-All}$$

$$\frac{\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2}{\Gamma \vdash \mathrm{Fun}\,(X{:}K_1)\,T_2 \in K_1 \rightarrow K_2} \tag{K-OAbs}$$

$$\frac{\Gamma \vdash S \in K_1 \rightarrow K_2 \qquad \Gamma \vdash T \in K_1}{\Gamma \vdash S\,T \in K_2} \tag{K-OApp}$$

$$\frac{\Gamma \vdash \mathrm{ok} \qquad \text{for each } i,\ \Gamma \vdash T_i \in K}{\Gamma \vdash \bigwedge^K [T_1..T_n] \in K} \tag{K-Meet}$$

### 3.2.3 Subtyping

The rules defining the subtype relation are a natural extension of familiar calculi of bounded quantification. Aside from some extra well-formedness conditions, the rules S-Trans, S-TVar, S-Arrow, and S-All are the same as in the usual, second-order case. Rules S-OAbs and S-OApp extend the subtype relation pointwise to kinds other than $\star$.

$F^\omega$'s rule of type conversion (if $\Gamma \vdash e \in T$ and $T =_\beta T'$ then $\Gamma \vdash e \in T'$) is captured here as the subtyping rule S-Conv, which also gives reflexivity as a special case.

The rule S-All compares the bodies of two quantifiers assuming a trivial upper bound for the variable $X$ instead of the usual assumption $X{\leq}T_1{:}K_1$ [CG92]. This variant, proposed by Castagna, is powerful enough for our purposes and has much better syntactic properties [CP94].

Rules S-Meet-G and S-Meet-LB specify that an intersection of a set of types is their order-theoretic greatest lower bound. S-Meet-Arrow, S-Meet-All, S-Meet-OAbs, and S-Meet-OApp allow intersections to distribute with the other operators.

As usual in systems with intersection types, the distributive laws S-Meet-Arrow through S-Meet-OApp are actually equivalences: for each constructor, the opposite direction can be proved using S-Meet-LB, S-Meet-G, the subtyping law for that constructor, and Proposition 3.3.6.

$$\frac{\Gamma \vdash S \in K \qquad \Gamma \vdash T \in K' \qquad S =_\beta T}{\Gamma \vdash S \leq T} \tag{S-Conv}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U} \tag{S-Trans}$$

$$\frac{\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash \mathrm{ok}}{\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash X \leq T} \tag{S-TVar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2 \qquad \Gamma \vdash S_1 {\rightarrow} S_2 \in \star}{\Gamma \vdash S_1 {\rightarrow} S_2 \leq T_1 {\rightarrow} T_2} \tag{S-Arrow}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma,\, X{\leq}\top^{K_1}{:}K_1 \vdash S_2 \leq T_2 \qquad \Gamma \vdash \mathrm{All}\,(X{\leq}S_1{:}K_1)\, S_2 \in \star}{\Gamma \vdash \mathrm{All}\,(X{\leq}S_1{:}K_1)\, S_2 \leq \mathrm{All}\,(X{\leq}T_1{:}K_1)\, T_2} \tag{S-All}$$

$$\frac{\Gamma,\, X{\leq}\top^K{:}K \vdash S \leq T}{\Gamma \vdash \mathrm{Fun}\,(X{:}K)\, S \leq \mathrm{Fun}\,(X{:}K)\, T} \tag{S-OAbs}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash S\,U \in K}{\Gamma \vdash S\,U \leq T\,U} \tag{S-OApp}$$

$$\frac{\text{for each } i,\ \Gamma \vdash S \leq T_i \qquad \Gamma \vdash S \in K}{\Gamma \vdash S \leq \bigwedge^K [T_1..T_n]} \tag{S-Meet-G}$$

$$\frac{\Gamma \vdash \bigwedge^K[T_1..T_n] \in K}{\Gamma \vdash \bigwedge^K[T_1..T_n] \leq T_i} \qquad \text{(S-Meet-LB)}$$

$$\frac{\Gamma \vdash S \to \bigwedge^\star[T_1..T_n] \in \star}{\Gamma \vdash \bigwedge^\star[S{\to}T_1 \ .. \ S{\to}T_n] \leq S \to \bigwedge^\star[T_1..T_n]} \qquad \text{(S-Meet-Arrow)}$$

$$\frac{\Gamma \vdash \text{All}\,(X{\leq}S{:}K) \ \bigwedge^\star[T_1..T_n] \in \star}{\Gamma \vdash \bigwedge^\star[\text{All}\,(X{\leq}S{:}K)\,T_1 \ .. \ \text{All}\,(X{\leq}S{:}K)\,T_n] \leq \text{All}\,(X{\leq}S{:}K) \ \bigwedge^\star[T_1..T_n]} \qquad \text{(S-Meet-All)}$$

$$\frac{\Gamma \vdash \text{Fun}\,(X{:}K_1) \ \bigwedge^{K_2}[T_1..T_n] \in K}{\Gamma \vdash \bigwedge^K[\text{Fun}\,(X{:}K_1)\,T_1 \ .. \ \text{Fun}\,(X{:}K_1)\,T_n] \leq \text{Fun}\,(X{:}K_1) \ \bigwedge^{K_2}[T_1..T_n]} \qquad \text{(S-Meet-OAbs)}$$

$$\frac{\Gamma \vdash \bigwedge^{K_1}[T_1..T_n]\,U \in K}{\Gamma \vdash \bigwedge^K[T_1\,U \ .. \ T_n\,U] \leq \bigwedge^{K_1}[T_1..T_n]\,U} \qquad \text{(S-Meet-OApp)}$$

### 3.2.4 Term Formation

Except for T-Meet and T-For, the term formation rules are precisely those of the second-order calculus of bounded quantification. T-For provides for typechecking under any of a set of alternate assumptions. For each $S_i$, the type derived for the instance of the body $e$ when $X$ is replaced by $S_i$ is a valid type of the for expression itself. The T-Meet rule can then be used to collect these separate typings into a single intersection. Type-theoretically, T-Meet is the introduction rule for the $\wedge$ constructor; the corresponding elimination rule need not be given explicitly, since it follows from T-Subsumption and S-Meet-LB.

$$\frac{\Gamma_1, \ x{:}T, \ \Gamma_2 \vdash \text{ok}}{\Gamma_1, \ x{:}T, \ \Gamma_2 \vdash x \in T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \ x{:}T_1 \vdash e \in T_2}{\Gamma \vdash \text{fun}\,(x{:}T_1)\,e \in T_1{\to}T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash f \in T_1{\to}T_2 \qquad \Gamma \vdash a \in T_1}{\Gamma \vdash f\,a \in T_2} \qquad \text{(T-App)}$$

$$\frac{\Gamma, \ X{\leq}T_1{:}K_1 \vdash e \in T_2}{\Gamma \vdash \text{fun}\,(X{\leq}T_1{:}K_1)\,e \in \text{All}\,(X{\leq}T_1{:}K_1)\,T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash f \in \text{All}\,(X{\leq}T_1{:}K_1)\,T_2 \qquad \Gamma \vdash S \leq T_1}{\Gamma \vdash f\,S \in T_2[X{\leftarrow}S]} \qquad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash e[X{\leftarrow}S] \in T \qquad S \in \{S_1..S_n\}}{\Gamma \vdash \text{for}\,(X \in S_1..S_n)\,e \in T} \qquad \text{(T-For)}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \text{for each } i,\ \Gamma \vdash e \in T_i}{\Gamma \vdash e \in \bigwedge^*[T_1..T_n]} \tag{T-Meet}$$

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \tag{T-Subsumption}$$

## 3.3 Basic Properties

This section establishes a number of structural properties of the definition of $F^\omega_\wedge$. Except where noted, the proofs proceed by structural induction, and are straightforward when performed in the order in which they appear.

**Lemma 3.3.1** *If $\Gamma \vdash \Sigma$ and $\Gamma_1$ is a prefix of $\Gamma$, then $\Gamma_1 \vdash ok$ as a subderivation. Moreover, except for the case $\Gamma_1 \equiv \Gamma$ and $\Sigma \equiv ok$, the subderivation is strictly shorter.*

**Lemma 3.3.2 (Free variables lemma)**

1. *$\Gamma \vdash T \in K$ implies $FTV(T) \subseteq dom(\Gamma)$.*

2. *If $\Gamma \vdash ok$, then each variable or type variable in $dom(\Gamma)$ is declared only once.*

**Lemma 3.3.3 (Syntax-directedness of context judgements)**

1. *If $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash ok$, then $\Gamma_1 \vdash T \in K$ by a proper subderivation.*

2. *If $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash ok$, then $\Gamma_1 \vdash T \in \star$ by a proper subderivation.*

**Lemma 3.3.4 (Weakening/Permutation)** *Let $\Gamma$ and $\Gamma'$ be contexts such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash ok$. Then $\Gamma \vdash \Sigma$ implies $\Gamma' \vdash \Sigma$.*

**Proof:** By induction on the lenght of a derivation of $\Gamma \vdash \Sigma$.

K-All — We are given that $\Gamma \vdash T_1 \in K_1$ and that $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$. Applying K-Meet to $\Gamma' \vdash$ ok we obtain $\Gamma' \vdash \top^{K_1} \in K_1$; we can assume, without loss of generality, that $X \notin$ dom$(\Gamma')$. Then, by C-TVar, $\Gamma',\ X{\leq}\top^{K_1}{:}K_1 \vdash$ ok. Now, by induction hypothesis, $\Gamma' \vdash T_1 \in K_1$ and $\Gamma',\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$, and applying K-All the result follows.

T-Abs — We are given that $\Gamma,\ x{:}T_1 \vdash e \in T_2$. By Lemma 3.3.1 there exists a proper subderivation of $\Gamma,\ x{:}T_1 \vdash$ ok; by Lemma 3.3.3, there is a yet shorter subderivation of $\Gamma \vdash T_1 \in \star$. Now we can apply the induction hypothesis to obtain $\Gamma' \vdash T_1 \in \star$. As before, we can assume $x \notin$ dom$(\Gamma')$; by C-Var, $\Gamma',\ x{:}T_1 \vdash$ ok. By the induction hypothesis we have $\Gamma',\ x{:}T_1 \vdash e \in T_2$, and applying T-Abs yields the desired result.

Other cases If $\Sigma \equiv$ ok there is nothing to prove. K-TVAR, S-TVAR, T-MEET and T-VAR applying the corresponding rule to $\Gamma' \vdash$ ok. K-OABS, S-ALL, and S-OABS similar to K-ALL. T-TABS similar to T-ABS. All the other cases follow by straightforward application of the induction hypothesis.

**Lemma 3.3.5 (Context, kind, and term strengthening)**

1. *If* $\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash ok$ *and* $X \notin FTV(\Gamma_2)$, *then* $\Gamma_1,\, \Gamma_2 \vdash ok$.
2. *If* $\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash S \in K'$ *and* $X \notin FTV(\Gamma_2) \cup FTV(S)$, *then* $\Gamma_1,\, \Gamma_2 \vdash S \in K'$.
3. *If* $\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash \Sigma$ *and* $x \notin FV(\Sigma)$, *then* $\Gamma_1,\, \Gamma_2 \vdash \Sigma$.

*Moreover, the derivations of the conclusions are strictly shorter than the derivation of the premises.*

**Proof:** 1 and 2 are proved by simultaneous induction on the length of derivations. 3 is proved by induction on the derivation of $\Gamma_1,\, x{:}T,\, \Gamma_2 \vdash \Sigma$.

1. C-EMPTY Vacuously true.

   C-VAR By part 2 of the induction hypothesis and C-VAR.

   C-TVAR $\Gamma_2 \equiv \emptyset$ The result follows from Lemma 3.3.1.
   $\Gamma_2 \not\equiv \emptyset$ By part 2 of the induction hypothesis and C-TVAR.

2. K-TVAR By part 1 of the induction hypothesis and K-TVAR.

   K-ARROW By part 2 of the induction hypothesis and K-ARROW.

   K-ALL We are given $\Gamma_1, X{\leq}T{:}K,\, \Gamma_2 \vdash T_1 \in K_1$, and $\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2,\, Y{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$, and $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(\mathrm{All}(Y{\leq}T_1{:}K_1)\,T_2)$. Observe that, by Lemma 3.3.2, $X \not\equiv Y$. Then $X \notin \mathrm{FTV}(\Gamma_2) \cup \mathrm{FTV}(T_1)$ and $X \notin \mathrm{FTV}(\Gamma_2,\, Y{\leq}\top^{K_1}{:}K_1) \cup \mathrm{FTV}(T_2)$. Now, applying part 2 of the induction hypothesis and K-ALL the result follows.

   K-OABS Similar to the case K-ALL.

   K-OAPP By part 2 of the induction hypothesis and K-OAPP.

   K-MEET By parts 1 and 2 of the induction hypothesis and K-OAPP.

3. Except for the cases we will consider below and the case for C-EMPTY, which is trivially true, the result follows by straightforward application of part 3 of the induction hypothesis and the corresponding rule in each case.

   C-VAR $\Gamma_2 \equiv \emptyset$ The result follows by Lemma 3.3.1.
   $\Gamma_2 \not\equiv \emptyset$ By part 3 of the induction hypothesis and C-VAR.

   T-ABS Using Lemma 3.3.2, part 3 of the induction hypothesis, and T-ABS.

   T-FOR Using that $\mathrm{FV}(e) = \mathrm{FV}(e[X{\leftarrow}S])$, part 3 of the induction hypothesis, and T-FOR.

**Proposition 3.3.6 (Syntax-directedness of kinding)**

1. $\Gamma \vdash X \in K$ *implies* $\Gamma \equiv \Gamma_1, X{\leq}T{:}K, \Gamma_2$ *for some* $\Gamma_1$, $T$, *and* $\Gamma_2$.
2. $\Gamma \vdash T_1{\rightarrow}T_2 \in K$ *implies* $K \equiv \star$ *and* $\Gamma \vdash T_1, T_2 \in \star$.
3. $\Gamma \vdash \mathit{All}(X{\leq}T_1{:}K_1)\, T_2 \in K$ *implies* $K \equiv \star$ *and* $\Gamma \vdash T_1 \in K_1$ *and* $\Gamma, X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$.
4. $\Gamma \vdash \mathit{Fun}(X{:}K_1)\, T_2 \in K$ *implies* $K \equiv K_1{\rightarrow}K_2$ *and* $\Gamma, X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K_2$, *for some* $K_2$.
5. $\Gamma \vdash S\,T \in K$ *implies* $\Gamma \vdash S \in K'{\rightarrow}K$ *and* $\Gamma \vdash T \in K'$, *for some* $K'$.
6. $\Gamma \vdash \bigwedge^K[T_1..T_n] \in K'$ *implies* $K \equiv K'$ *and* $\Gamma \vdash \mathit{ok}$ *and* $\Gamma \vdash T_i \in K$ *for each* $i$.

*Moreover, the proofs of the consequents are all strictly shorter than the antecedents.*

**Proof:** By inspection.

**Lemma 3.3.7 (Uniqueness of kinds)** $\Gamma \vdash T \in K$ *and* $\Gamma \vdash T \in K'$, *then* $K \equiv K'$.

**Lemma 3.3.8 (Type substitution)** *Let* $\Gamma_1 \vdash T \in K_U$. *Then*

1. *If* $\Gamma_1, X{\leq}U{:}K_U, \Gamma_2 \vdash S \in K_S$, *then* $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash S[X{\leftarrow}T] \in K_S$.
2. *If* $\Gamma_1, X{\leq}U{:}K_U, \Gamma_2 \vdash \mathit{ok}$, *then* $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash \mathit{ok}$.

**Proof:** By simultaneous induction on derivations of the premises. The proof of part 2 is straightforward using part 1 of the induction hypothesis. We consider the details of the proof of 1. The cases K-Arrow, K-All, K-OAbs, and K-OApp follow by straightforward application of part 1 of the induction hypothesis and the corresponding rule, while the case of K-Meet also uses part 2 of the induction hypothesis. Now take the case of K-TVar, where $S \equiv Y$ for some variable $Y$. By Proposition 3.3.6(1) $Y{\leq}T_Y{:}K_S \in (\Gamma_1, X{\leq}U{:}K_U, \Gamma_2)$ for some $T_Y$. There are three cases to consider.

| | |
|---|---|
| $Y{\leq}T_Y{:}K_S \in \Gamma_1$ | Then we also have $Y{\leq}T_Y{:}K_S \in (\Gamma_1, \Gamma_2[X{\leftarrow}T])$. By part 2 of the induction hypothesis, $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash$ ok. Applying K-TVar, we get $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash Y \in K_S$. |
| $Y{\leq}T_Y{:}K_S \equiv X{\leq}U{:}K_U$ | We know that $\Gamma_1 \vdash T \in K_S \equiv K_U$. From the premise of K-TVar and part 2 of the induction hypothesis, we have $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash$ ok. The result follows by weakening (3.3.4). |
| $Y{\leq}T_Y{:}K_S \in \Gamma_2$ | Then we have $Y{\leq}T_Y[X{\leftarrow}T]{:}K_S \in (\Gamma_1, \Gamma_2[X{\leftarrow}T])$. By part 2 of the induction hypothesis, $\Gamma_1, \Gamma_2[X{\leftarrow}T] \vdash$ ok, from which the result follows from K-TVar. |

**Lemma 3.3.9 (Kind invariance under type conversion)** *If* $\Gamma \vdash S \in K_S$ *and* $\Gamma \vdash T \in K_T$*, with* $S =_\beta T$*, then* $K_S \equiv K_T$*.*

**Proof sketch:** By induction on the definition of $\beta$-conversion, it is easy to see that it suffices to show the statement for a one-step reduction $S \rightarrow_\beta T$. This is proved by induction on the structure of $S$. The interesting case is when $S$ is a $\beta$-redex and $T$ its reduct: $S \equiv (\mathrm{Fun}\,(X{:}K)\,S_1)S_2$ and $T \equiv S_1[X \leftarrow S_2]$. By Proposition 3.3.6 (parts 5 and then 4), $\Gamma,\ X {\leq} \top^K{:}K \vdash S_1 \in K_S$ and $\Gamma \vdash S_2 \in K$. Now, by the type substitution lemma (3.3.8), $\Gamma \vdash S_1[X \leftarrow S_2] \in K_S$ and by uniqueness of kinds, it follows that $K_S \equiv K_T$.

**Lemma 3.3.10 (Context modification)** *If* $\Gamma_1 \vdash U' \in K$ *and* $\Sigma$ *is either ok or* $T \in K'$*, then* $\Gamma_1,\ X {\leq} U{:}K,\ \Gamma_2 \vdash \Sigma$ *implies* $\Gamma_1,\ X {\leq} U'{:}K,\ \Gamma_2 \vdash \Sigma$*.*

**Proposition 3.3.11 (Well-kindedness of subtyping)** *If* $\Gamma \vdash S \leq T$*, then* $\Gamma \vdash S \in K$ *and* $\Gamma \vdash T \in K$ *for some* $K$*.*

**Proof:** By induction on the derivation of $\Gamma \vdash S \leq T$.

S-Conv: We are given that $\Gamma \vdash S \in K$ and $\Gamma \vdash T \in K'$ and $S =_\beta T$. By Lemma 3.3.9, $K \equiv K'$.

S-Trans: By induction hypothesis and uniqueness of kinds (3.3.7).

S-TVar: We are given that $\Gamma_1,\ X {\leq} T{:}K,\ \Gamma_2 \vdash \mathrm{ok}$. By K-TVar we get $\Gamma_1,\ X {\leq} T{:}K,\ \Gamma_2 \vdash X \in K$. Moerover, by Lemma 3.3.3 we have $\Gamma_1 \vdash T \in K$, and by Lemma 3.3.4, $\Gamma_1,\ X {\leq} T{:}K,\ \Gamma_2 \vdash T \in K$.

S-Arrow: We are given $\Gamma \vdash T_1 \leq S_1$ and $\Gamma \vdash S_2 \leq T_2$ and $\Gamma \vdash S_1 {\rightarrow} S_2 \in \star$. By Proposition 3.3.6 $\Gamma \vdash S_1, S_2 \in \star$. Further, by the induction hypothesis together with uniqueness of kinds (3.3.7), we have $\Gamma \vdash T_1, T_2 \in \star$. Finally, applying K-Arrow, the result follows.

S-All: We are given that $\Gamma \vdash T_1 \leq S_1$ and $\Gamma,\ X {\leq} \top^{K_1}{:}K_1 \vdash S_2 \leq T_2$ and also $\Gamma \vdash \mathrm{All}\,(X {\leq} S_1{:}K_1)\,S_2 \in \star$. By Proposition 3.3.6 $\Gamma \vdash S_1 \in K_1$ and $\Gamma,\ X {\leq} \top^{K_1}{:}K_1 \vdash S_2 \in \star$. Then, applying the induction hypothesis and Lemma 3.3.7, we obtain $\Gamma \vdash T_1 \in K_1$ and $\Gamma,\ X {\leq} \top^{K_1}{:}K_1 \vdash T_2 \in \star$ from which, by K-All, the result follows.

S-OAbs: By induction hypothesis and K-OAbs.

S-OApp: Similar to S-All.

S-Meet-G: Using the induction hypothesis, Lemma 3.3.7, and K-Meet.

S-Meet-LB: We are given $\Gamma \vdash \bigwedge^K [T_1..T_n] \in K$, which, by Proposition 3.3.6, implies $\Gamma \vdash T_i \in K$ for each $i$.

S-Meet-All: We are given $\Gamma \vdash \mathrm{All}\,(X {\leq} S{:}K)\ \bigwedge^\star [T_1..T_n] \in \star$. We have to consider two cases.

$n = 0$ By Lemma 3.3.1, it follows that $\Gamma \vdash \mathrm{ok}$, and by K-MEET, $\Gamma \vdash \top^{\star} \in \star$.

$n \neq 0$ By Proposition 3.3.6 we have $\Gamma \vdash S \in K$ and $\Gamma,\ X{\leq}\top^{K}{:}K \vdash T_i \in \star$ for every $i$, which, by K-ALL, yields $\Gamma \vdash \mathrm{All}\,(X{\leq}S{:}K)\,T_i \in \star$ for each $i$. Finally, applying K-MEET the result follows.

Other cases — S-MEET-ARROW, S-MEET-OAPP, and S-MEET-OABS are proved like S-MEET-ALL.

**Proposition 3.3.12 (Well-kindedness of typing)** *If $\Gamma \vdash e \in T$, then $\Gamma \vdash T \in \star$.*

**Proof:** By induction on the derivation of $\Gamma \vdash e \in T$.

T-VAR — We are given $\Gamma_1,\ x{:}T,\ \Gamma_2 \vdash \mathrm{ok}$. The result follows by Lemma 3.3.3 and Lemma 3.3.4.

T-ABS — We are given $\Gamma,\ x{:}T_1 \vdash e \in T_2$. By induction hypothesis $\Gamma,\ x{:}T_1 \vdash T_2 \in \star$. By Lemma 3.3.5, it follows that $\Gamma \vdash T_2 \in \star$. On the other hand, by Lemmas 3.3.1 and 3.3.3, $\Gamma \vdash T_1 \in \star$. Hence, K-ARROW yields $\Gamma \vdash T_1{\rightarrow}T_2 \in \star$.

T-APP — By induction hypothesis on $\Gamma \vdash f \in T_1{\rightarrow}T$ and Proposition 3.3.6.

T-TABS — We are given $\Gamma,\ X{\leq}T_1{:}K_1 \vdash e \in T_2$. By induction hypothesis $\Gamma,\ X{\leq}T_1{:}K_1 \vdash T_2 \in \star$ and, by the Context-Modification Lemma (3.3.10), $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash e \in T_2$. Hence, applying K-ALL, we obtain $\Gamma \vdash \mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2 \in \star$.

T-TAPP — We know that $\Gamma \vdash f \in \mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2$ and also $\Gamma \vdash S \leq T_1$. By the induction hypothesis, $\Gamma \vdash \mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2 \in \star$ and, by Proposition 3.3.6, $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in \star$ and $\Gamma \vdash T_1 \in K_1$. Now, by the well-kindedness of suptyping (3.3.11) and uniqueness of kinds (3.3.7), we have $\Gamma \vdash S \in K_1$. Then, by the type substitution lemma (3.3.8),$\Gamma \vdash T_2[X{\leftarrow}S] \in \star$.

T-FOR — By induction hypothesis.

T-MEET — We are given that $\Gamma \vdash \mathrm{ok}$ and that $\Gamma \vdash e \in T_i$ for each $i$. We have to consider two cases.

$n = 0$ Applying K-MEET to $\Gamma \vdash \mathrm{ok}$ we obtain $\Gamma \vdash \top^{\star} \in \star$.

$n \neq 0$ By the induction hypothesis, $\Gamma \vdash T_i \in \star$ for every $i$ and, applying K-MEET, the result follows.

T-SUBSUMPTION — By induction hypothesis using Proposition 3.3.11 and Lemma 3.3.7.

# 4 Semantics

Among simplest models for typed $\lambda$-calculi are those based on partial equivalence relations (PERs). A model in this style is essentially untyped: terms are interpreted by erasing all type information and interpreting the resulting pure $\lambda$-term as an element of the model. A type, in this setting, is just a subset of the model along with an appropriate notion of equivalence of elements. Coercions between types are interpreted as inclusion of PERs.

Our PER model for $F^{\omega}_{\wedge}$ extends the model of $F_{\wedge}$ given in [Pie91], which is based on Bruce and Longo's model for $F_{\leq}$ [BL90]. The usual interpretation of a quantified type $\mathrm{All}\,(X)\,T$ in a PER model is the PER-indexed intersection of all possible instances of $T$. Bruce and Longo refined this definition to interpret $\mathrm{All}\,(X{\leq}S)\,T$ as the intersection of all the instances of $T$ where $X$ is interpreted as a sub-PER of the interpretation of $S$. This intuition also serves for intersection types: $\bigwedge^{\star}[T_1..T_n]$ is interpreted as the intersection of the PERs interpreting each of the $T_i$'s. We generalize this model to $\omega$-order polymorphism (and subtyping) by interpreting type operators as functions over PERs.

To deal correctly with intersection types, we need to make one significant technical departure here from PER models of ordinary bounded quantification: instead of allowing the elements of our PERs to be drawn from the carrier of an arbitrary partial combinatory algebra $\mathcal{D}$, we require that $\mathcal{D}$ be a *total* combinatory algebra. This restriction is needed to validate nullary instances of the distributive law S-Meet-Arrow, which have the form $\Gamma \vdash \top^{\star} \leq S \to \top^{\star}$. For example, let $S = \top^{\star}$. The empty intersection $\top^{\star}$ is interpreted by the everywhere-defined PER, i.e., $[\![\top^{\star}]\!]$ relates every $m$ to itself. To validate the distributivity law, it must therefore be the case that $[\![\top^{\star} \to \top^{\star}]\!]$ relates every element to itself. But this will only be true if the application of any element to any other element is defined. This observation is due to QingMing Ma [personal communication, 1991].

Cardelli and Longo [CL91] and Bruce and Mitchell [BM92] have given related models for variants of $F^{\omega}$ including subtyping, but without intersections.

The notation and fundamental definitions used here are based on papers of Bruce and Longo [BL90], Freyd, Mulry, Rosolini, and Scott [FMRS90], and others. A helpful basic reference for PER models of second-order $\lambda$-calculi is [Mit90b]; also see [BMM90] for more general discussion of second-order models and [Bar84, HS86] for general discussion of combinatory models.

## 4.1 Total Combinatory Algebras

A *total combinatory algebra* is a tuple $\mathcal{D} = \langle D, \cdot, k, s\rangle$ comprising a set $D$ of *elements*, an *application function* $\cdot$ with type $D \to (D \to D)$, and distinguished elements $k, s \in D$ such that, for all $d_1, d_2, d_3 \in D$,

$$k \cdot d_1 \cdot d_2 = d_1$$
$$s \cdot d_1 \cdot d_2 \cdot d_3 = (d_1 \cdot d_3) \cdot (d_2 \cdot d_3).$$

Throughout this section, we work with a fixed, but unspecified, total combinatory algebra $\mathcal{D}$. (C.f. [Sco76] for examples.)

The set of *pure λ-terms* is defined by the following grammar:

$$M ::= x \mid \mathrm{fun}\,(x)\,M \mid M_1\ M_2$$

The set of *combinator terms* is:

$$C ::= x \mid C_1\ C_2 \mid K \mid S$$

The *bracket abstraction* of a combinator term $C$ with respect to a variable $x$, written $\mathrm{fun}^\star\,(x)\,C$, is defined as follows:

$$\begin{array}{lcll}
\mathrm{fun}^\star\,(x)\,C & = & K\,C & \text{when } x \notin \mathrm{FV}(C) \\
\mathrm{fun}^\star\,(x)\,x & = & S\,K\,K & \\
\mathrm{fun}^\star\,(x)\,C_1\,C_2 & = & S\,(\mathrm{fun}^\star\,(x)\,C_1)\,(\mathrm{fun}^\star\,(x)\,C_2) & \text{when } x \in \mathrm{FV}(C_1\,C_2)
\end{array}$$

The *combinator translation* of a pure λ-term $M$, written $|M|$, is defined as follows:

$$\begin{array}{lcl}
|x| & = & x \\
|\mathrm{fun}\,(x)\,M| & = & \mathrm{fun}^\star\,(x)\,|M| \\
|M_1\,M_2| & = & |M_1|\,|M_2|
\end{array}$$

A *term environment* $\eta$ is a finite function from term variables to elements of $D$. When $x \notin \mathrm{dom}(\eta)$, we write $\eta[x{\leftarrow}d]$ for the environment that maps $x$ to $d$ and agrees with $\eta$ everywhere else. We write $\eta\backslash x$ for the environment like $\eta$ except that $\eta(x)$ is undefined; $\eta\backslash\Gamma$ is like $\eta$ but undefined on all the variables in $\mathrm{dom}(\Gamma)$. We say that $\eta'$ *extends* $\eta$ when $\mathrm{dom}(\eta) \subseteq \mathrm{dom}(\eta')$ and $\eta$ and $\eta'$ agree on $\mathrm{dom}(\eta)$.

Let $C$ be a combinator term and $\eta$ a term environment such that $\mathrm{FV}(C) \subseteq \mathrm{dom}(\eta)$. Then the *interpretation* of $C$ under $\eta$, written $[\![C]\!]_\eta$, is defined as follows:

$$\begin{array}{lcl}
[\![x]\!]_\eta & = & \eta(x) \\
[\![C_1\,C_2]\!]_\eta & = & [\![C_1]\!]_\eta \cdot [\![C_2]\!]_\eta \\
[\![K]\!]_\eta & = & k \\
[\![S]\!]_\eta & = & s.
\end{array}$$

**Lemma 4.1.1**

1. *If $\eta'$ extends $\eta$ and $FV(C) \subseteq dom(\eta)$, then $[\![C]\!]_\eta = [\![C]\!]_{\eta'}$.*

2. $[\![\mathit{fun}^\star\,(x)\,C]\!]_\eta \cdot m = [\![C]\!]_{\eta[x\leftarrow m]}$.

**Proof:** Standard.

## 4.2 Higher-Order Partial Equivalence Relations

A *partial equivalence relation* (PER) on a combinatory algebra $\mathcal{D}$ is a symmetric and transitive relation $A$ on $D$. We write $m\ \{A\}\ n$ when $A$ relates $m$ and $n$. The *domain* of $A$, written $\mathrm{dom}(A)$, is the set $\{n \mid n\ \{A\}\ n\}$. (Note that $m\ \{A\}\ n$ implies $m \in \mathrm{dom}(A)$.) We write **PER** for the class of all PERs.

If $A$ and $B$ are relations, then $A \to B$ is the relation where $m\ \{A \to B\}\ n$ iff, for all $p, q \in D$, $p\ \{A\}\ q$ $m{\cdot}p\ \{B\}\ n{\cdot}q$. It is not hard to show that $A \to B$ is a PER when $A$ and $B$ are PERs, and that the intersection of any set of PERs is a PER.

To interpret type operators, we need to consider not only PERs, but arbitrary function spaces built on PER. An element of such a function space (including, as a special case, an element of PER itself), is called a *higher-order PER* (HOPER). The interpretation of a kind $K$ is a suitable space of HOPERs:

$$\begin{array}{lcl} [\![\star]\!] & = & \mathbf{PER} \\ [\![K_1 \to K_2]\!] & = & [\![K_1]\!] \to [\![K_2]\!]. \end{array}$$

We generalize the familiar graph-inclusion of relations to HOPERs as follows:

$$\begin{array}{lll} A \subseteq^\star B & \text{iff} & A, B \in [\![\star]\!] \text{ and } m\ \{A\}\ n \text{ implies } m\ \{B\}\ n \text{ for all } m, n \in D; \\ A \subseteq^{K_1 \to K_2} B & \text{iff} & A, B \in [\![K_1 \to K_2]\!] \text{ and } A\,P \subseteq^{K_2} B\,P \text{ for all } P \in [\![K_1]\!]. \end{array}$$

Let $\{A_i \in [\![K]\!]\}_{i \in I}$ be a set of HOPERs indexed by a set $I$. Then $\bigcap^K_{i\in I} A_i$ is the HOPER defined by

$$\begin{array}{lll} m\ \{\bigcap^\star_{i\in I} A_i\}\ n & \text{iff} & \text{for every } i,\ m\ \{A_i\}\ n \\ \bigcap^{K_1 \to K_2}_{i\in I} A_i & = & \lambda P {\in} [\![K_1]\!].\ \bigcap^{K_2}_{i\in I} A_i\,P \end{array}$$

**Lemma 4.2.1**

1. *Each* $\subseteq^K$ *is transitive.*
2. *If* $A_j \in [\![K]\!]$ *for each* $j{\in}I$*, then* $\bigcap^K_{i\in I} A_i \in [\![K]\!]$.
3. *If* $A \subseteq^K B_j$ *for each* $j$*, then* $A \subseteq^K \bigcap^K_{i\in I} B_i$.
4. $\bigcap^K_{i\in I} A_i \subseteq^K A_j$ *for each* $j$.
5. $\bigcap^\star_{i\in I} A {\to} B_i \subseteq^\star A \to \bigcap^\star_{i\in I} B_i$.
6. $\bigcap^\star_{i\in I} \bigcap^\star_{P \subseteq^K A} B_i \subseteq^\star \bigcap^\star_{P \subseteq^K A} \bigcap^\star_{i\in I} B_i$.
7. $\bigcap^{K_1 \to K_2}_{i\in I} \lambda P {\in} [\![K_1]\!].\ B_i\,P \subseteq^{K_1 \to K_2} \lambda P {\in} [\![K_1]\!].\ \bigcap^{K_2}_{i\in I} B_i\,P$*, when each* $B_i \in [\![K_1 {\to} K_2]\!]$.
8. $\bigcap^{K_2}_{i\in I} A\ B_i \subseteq^{K_2} A\ (\bigcap^{K_1 \to K_2}_{i\in I} B_i)$.

*Indeed, in cases 4 through 8 the inclusions are equalities.*

**Proof:** Straightforward.

Each collection $[\![K]\!]$ of HOPERs has a maximal element under the ordering $\subseteq^K$. This element is written $\top^K$ and can be calculated as follows:

- $\top^\star$ is the total relation on $\mathcal{D}$.
- $\top^{K_1 \to K_2} = \lambda P {\in} [\![K_1]\!].\ \top^{K_2}$.

**Fact 4.2.2** *Let $A \in [\![K]\!]$. Then:*

1. $A \subseteq^K \top^K$.
2. $\top^K \subseteq^K A$ *implies* $A = \top^K$.

## 4.3 HOPER Interpretation of $F^\omega_\wedge$

An *environment* $\eta$ is a finite function from type variables to HOPERs and from term variables to elements of $D$. The notations for environment extension, restriction, and agreement are carried over from term environments. By an abuse of notation, type environments are used in place of term environments from now on.

The *erasure* of an $F^\omega_\wedge$ term $e$, written erase$(e)$, is the pure $\lambda$-term defined as follows:

$$
\begin{array}{lcl}
\mathrm{erase}(x) & = & x \\
\mathrm{erase}(\mathrm{fun}\,(x{:}T)\,e) & = & \mathrm{fun}\,(x)\,\mathrm{erase}(e) \\
\mathrm{erase}(e_1\ e_2) & = & \mathrm{erase}(e_1)\ \mathrm{erase}(e_2) \\
\mathrm{erase}(\mathrm{fun}\,(X{\leq}T{:}K)\,e) & = & \mathrm{erase}(e) \\
\mathrm{erase}(e\ T) & = & \mathrm{erase}(e) \\
\mathrm{erase}(\mathrm{for}\,(X{\in}T_1..T_n)\,e) & = & \mathrm{erase}(e).
\end{array}
$$

Let $\eta$ be a term environment and $e$ an expression such that $\mathrm{FV}(e) \subseteq \mathrm{dom}(\eta)$. Then the *interpretation* of $e$ under $\eta$, written $[\![e]\!]_\eta$, is $[\![|\mathrm{erase}(e)|]\!]_\eta$.

If $\eta$ is an environment and $T$ a type expression such that $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\eta)$, then the *interpretation* of $T$ under $\eta$, written $[\![T]\!]_\eta$, is the HOPER defined as follows:

$$
\begin{array}{lcl}
[\![X]\!]_\eta & = & \eta(X) \\
[\![T_1 \to T_2]\!]_\eta & = & [\![T_1]\!]_\eta \to [\![T_2]\!]_\eta \\
[\![\mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2]\!]_\eta & = & \bigcap^\star_{P \subseteq^{K_1} [\![T_1]\!]_\eta} [\![T_2]\!]_{\eta[X \leftarrow P]} \\
[\![\bigwedge^K[T_1..T_n]]\!]_\eta & = & \bigcap^K_{1 \leq i \leq n} [\![T_i]\!]_\eta \\
[\![S\,T]\!]_\eta & = & [\![S]\!]_\eta\ [\![T]\!]_\eta \\
[\![\mathrm{Fun}\,(X{:}K)\,T]\!]_\eta & = & \lambda P {\in} [\![K]\!].\ [\![T]\!]_{\eta[X \leftarrow P]}
\end{array}
$$

Of course, for some $\eta$'s and $T$'s, the interpretation $[\![T]\!]_\eta$ given by these equations is nonsensical. We write $[\![T]\!]_\eta{\downarrow}$ when $[\![T]\!]_\eta$ is well defined and $[\![T]\!]_\eta{\uparrow}$ otherwise. Lemma 4.3.1 will show that the interpretation of a well-kinded type is always well defined.

We say that an environment $\eta$ *satisfies* a context $\Gamma$, written $\eta \models \Gamma$, if $\mathrm{dom}(\eta) = \mathrm{dom}(\Gamma)$ and

1. $\Gamma \equiv \emptyset$; or

2. $\Gamma \equiv \Gamma_1,\ x{:}T$, where $\eta\backslash x$ satisfies $\Gamma_1$ and either $[\![T]\!]_{\eta\backslash x}\uparrow$ or $\eta(x) \in \mathrm{dom}([\![T]\!]_{\eta\backslash x})$; or

3. $\Gamma \equiv \Gamma_1,\ X{\leq}T{:}K$, where $\eta\backslash X$ satisfies $\Gamma_1$ and either $[\![T]\!]_{\eta\backslash X}\uparrow$ or $\eta(X) \subseteq^K [\![T]\!]_{\eta\backslash X}$.

Iterating the definition immediately yields that either $[\![T]\!]_{\eta\backslash\Gamma_2\backslash X}\uparrow$ or $[\![T]\!]_{\eta\backslash\Gamma_2\backslash X} \in [\![K]\!]$, whenever $\eta \models \Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2$. Also, note that if $\eta'$ extends $\eta$ and $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\eta)$, then either $[\![T]\!]_\eta\uparrow$ and $[\![T]\!]_{\eta'}\uparrow$, or else both are defined and $[\![T]\!]_\eta = [\![T]\!]_{\eta'}$.

**Lemma 4.3.1** *Let $T$ be a type, $\Gamma$ a context, and $\eta$ an environment such that $FTV(T) \subseteq dom(\eta)$ and $\eta \models \Gamma$. If $\Gamma \vdash T \in K$, then $[\![T]\!]_\eta\downarrow$ and $[\![T]\!]_\eta \in [\![K]\!]$.*

**Proof:** We need to check the desired result together with an additional fact:

1. If $\Gamma \vdash T \in K$, then $[\![T]\!]_\eta$ is defined and $[\![T]\!]_\eta \in [\![K]\!]$.

2. If $\Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2 \vdash \mathrm{ok}$, then $[\![T]\!]_{\eta\backslash\Gamma_2\backslash X}\downarrow$ and $[\![T]\!]_{\eta\backslash\Gamma_2\backslash X} \in [\![K]\!]$.

The two are proved by simultaneous induction on derivations. We give only the interesting cases; the rest follow by straightforward use of the induction hypothesis and simple properties of HOPERs.

1. K-TVar We are given that $\Gamma \equiv \Gamma_1,\ X{\leq}T{:}K,\ \Gamma_2$ and that $\Gamma \vdash \mathrm{ok}$. By part 2 of the induction hypothesis, $[\![T]\!]_{\eta\backslash\Gamma_2\backslash X}\downarrow$. By the definition of satisfaction, $(\eta\backslash\Gamma_2)(X) \subseteq^K [\![T]\!]_{\eta\backslash\Gamma_2\backslash X}$, which implies in particular that $\eta(X) \in [\![K]\!]$. By the definition of interpretation, $[\![X]\!]_\eta = \eta(X) \in [\![K]\!]$.

   K-All We are given that $K \equiv \star$ and $T \equiv \mathrm{All}(X{\leq}T_1{:}K_1)\,T_2$, and that $\Gamma \vdash T_1 \in K_1$ and $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash T_2 \in K$. The induction hypothesis yields $[\![T_1]\!]_\eta\downarrow$ and $[\![T_1]\!]_\eta \in [\![K_1]\!]$. Now, suppose $P \subseteq^{K_1} [\![T_1]\!]_\eta$. Since $P \subseteq^{K_1} \top^{K_1}$, the definition of satisfaction yeilds $\eta[X{\leftarrow}P] \models \Gamma,\ X{\leq}\top^{K_1}{:}K_1$. By the induction hypothesis, $[\![T_2]\!]_{\eta[X\leftarrow P]}\downarrow$ and $[\![T_2]\!]_{\eta[X\leftarrow P]} \in [\![\star]\!]$. Since PER is closed under intersections, $[\![T]\!]_\eta = \bigcap^\star_{P\subseteq^{K_1}[\![T_1]\!]_\eta} [\![T_2]\!]_{\eta[X\leftarrow P]} \in [\![\star]\!]$.

   K-OAbs Similar.

2. C-Var We are given that $\Gamma \equiv \Gamma',\ x{:}S$, where $\Gamma' \equiv \Gamma_1,\ X{\leq}T{:}K,\ \Gamma'_2$ and that $\Gamma' \vdash S \in \star$. By Lemma 3.3.1, $\Gamma' \vdash \mathrm{ok}$. By the definition of satisfaction, $\eta\backslash x \models \Gamma'$. By the induction hypothesis, the result follows.

   C-TVar The case where $\Gamma_2 \not\equiv \emptyset$ is similar to the previous case. When $\Gamma_2 \equiv \emptyset$, we are given that $\Gamma \equiv \Gamma_1,\ X{\leq}T{:}K$ and that $\Gamma_1 \vdash T \in K$. By the definition of satisfaction, $\eta\backslash X \models \Gamma'$. By the induction hypothesis, the result follows.

In order to prove the soundness of subtyping we need some technical results about substitution and $\beta$-conversion.

**Lemma 4.3.2** *Let $\eta$ be an environment with $X \notin dom(\eta)$ and such that $FTV(S[X \leftarrow T]) \subseteq dom(\eta)$ and $[\![S]\!]_{\eta[X \leftarrow [\![T]\!]_\eta]}\downarrow$. Then $[\![S[X \leftarrow T]]\!]_\eta = [\![S]\!]_{\eta[X \leftarrow [\![T]\!]_\eta]}$.*

**Proof:** By induction on the structure of S.

**Lemma 4.3.3** *Let $\eta$ be an environment such that $FTV(S) \subseteq dom(\eta)$. If $S =_\beta T$ and $[\![S]\!]_\eta\downarrow$, then $[\![S]\!]_\eta = [\![T]\!]_\eta$.*

**Proof:** By induction on the definition of $\beta$-conversion, it is easy to see that it suffices to show the statement for a one-step reduction $S \rightarrow_\beta T$. This is proved by induction on the structure of $S$. The only interesting case is when S is a $\beta$-redex and $T$ its reduct. Let $S \equiv (\mathrm{Fun}\,(X{:}K)\,T_2)T_1$ and $T \equiv T_1[X \leftarrow T_2]$. Then

$$\begin{array}{rcll} [\![(\mathrm{Fun}\,(X{:}K)\,T_2)T_1]\!]_\eta & = & (\lambda P \in [\![K]\!].\, [\![T_2]\!]_{\eta[X \leftarrow P]})[\![T_1]\!]_\eta & \text{by definition of } [\![\,-\,]\!]_\eta \\ & = & [\![T_2]\!]_{\eta[X \leftarrow [\![T_1]\!]_\eta]} & \\ & = & [\![T_2[X \leftarrow T_1]]\!]_\eta & \text{by Lemma 4.3.2.} \end{array}$$

Our main semantic results are the soundness of subtyping and typing.

**Theorem 4.3.4 (Soundness of subtyping)** *If $\Gamma \vdash S \leq T$ and $\Gamma \vdash S \in K$, and if $\eta \models \Gamma$, then $[\![S]\!]_\eta \subseteq^K [\![T]\!]_\eta$.*

**Proof:** The proof proceeds by induction on the structure of a derivation of $\Gamma \vdash S \leq T$. For the sake of readability, we often make implicit use of the fact that if $\Gamma \vdash S \leq T$ and $\Gamma \vdash S \in K$, then, by Proposition 3.3.11 and Lemmas 3.3.7 and 4.3.1, $[\![S]\!]_\eta$ and $[\![T]\!]_\eta$ are defined and belong to $[\![K]\!]$ whenever $\eta \models \Gamma$.

S-Conv — We are given that $S =_\beta T$. Then, by Lemma 4.3.3, $[\![S]\!]_\eta = [\![T]\!]_\eta$

S-Trans — By the induction hypothesis and the transitivity of $\subseteq^K$ (4.2.1).

S-TVar — We are given $\Gamma_1,\, X{\leq}T{:}K,\, \Gamma_2 \vdash \mathrm{ok}$. Now, $[\![X]\!]_\eta = \eta(X) \subseteq^K [\![T]\!]_{\eta\backslash X}$, because $\eta \models \Gamma$ and, as $\eta$ extends $\eta\backslash X$, $[\![T]\!]_{\eta\backslash X} = [\![T]\!]_\eta$.

S-Arrow — We are given $\Gamma \vdash T_1 \leq S_1$ and $\Gamma \vdash S_2 \leq T_2$, with $\Gamma \vdash S_1{\rightarrow}S_2 \in \star$. By the uniqueness of kinds (3.3.7), $K \equiv \star$. Now by the well-kindedness of subtyping (3.3.11) and syntax directedness of kinds (3.3.6) we have $\Gamma \vdash S_2, T_1 \in \star$. By the induction hypothesis, $[\![S_2]\!]_\eta \subseteq^\star [\![T_2]\!]_\eta$ and $[\![T_1]\!]_\eta \subseteq^\star [\![S_1]\!]_\eta$. Hence, by the covariance on the right and contravariance on the left of the function space constructor on PERs (easily verified from its definition), $[\![S_1]\!]_\eta{\rightarrow}[\![S_2]\!]_\eta \subseteq^\star [\![T_1]\!]_\eta{\rightarrow}[\![T_2]\!]_\eta$, i.e. $[\![S_1{\rightarrow}S_2]\!]_\eta \subseteq^\star [\![T_1{\rightarrow}T_2]\!]_\eta$.

S-All — We are given that $\Gamma \vdash T_1 \leq S_1$ and $\Gamma,\, X{\leq}\top^{K_1}{:}K_1 \vdash S_2 \leq T_2$ and that $\Gamma \vdash \mathrm{All}\,(X{\leq}S_1{:}K_1)\,S_2 \in \star$. Let $P \in [\![K_1]\!]_\eta$. Then $\eta[X \leftarrow P] \models \Gamma,\, X{\leq}\top^{K_1}{:}K_1$.

By Proposition 3.3.6, $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash S_2 \in \star$. Now, by the induction hypothesis, $[\![S_2]\!]_{\eta[X\leftarrow P]} \subseteq^{\star} [\![T_2]\!]_{\eta[X\leftarrow P]}$. By the well-kindedness of subtyping and syntax-directedness of kinds, $\Gamma \vdash T_1 \in K_1$. Applying the induction hypothesis we obtain $[\![T_1]\!]_\eta \subseteq^{K_1} [\![S_1]\!]_\eta$. Hence $\bigcap^{\star}_{P\subseteq^{K_1}[\![S_1]\!]_\eta} [\![S_2]\!]_{\eta[X\leftarrow P]} \subseteq^{\star} \bigcap^{\star}_{P\subseteq^{K_1}[\![T_1]\!]_\eta} [\![T_2]\!]_{\eta[X\leftarrow P]}$, i.e. $[\![\mathrm{All}\,(X{\leq}S_1{:}K_1)\,S_2]\!]_\eta \subseteq^{\star} [\![\mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2]\!]_\eta$.

S-OAbs — We are given $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash S \leq T$ and $\Gamma \vdash \mathrm{Fun}\,(X{:}K_1)\,S \in K$. By the syntax-directedness of kinding, $K \equiv K_1{\rightarrow}K_2$ and $\Gamma,\ X{\leq}\top^{K_1}{:}K_1 \vdash S \in K_2$ for some $K_2$. Let $P \in [\![K_1]\!]_\eta$. Then $\eta[X\leftarrow P] \models \Gamma,\ X{\leq}\top^{K_1}{:}K_1$. Now by the induction hypothesis, $[\![S]\!]_{\eta[X\leftarrow P]} \subseteq^{K_2} [\![T]\!]_{\eta[X\leftarrow P]}$. Then $\lambda P \in [\![K_1]\!].\ [\![S]\!]_{\eta[X\leftarrow P]} \subseteq^{K_1\rightarrow K_2} \lambda P {\in} [\![K_1]\!].\ [\![T]\!]_{\eta[X\leftarrow P]}$, i.e. $[\![\mathrm{Fun}\,(X{:}K_1)\,S]\!]_\eta \subseteq^{K_1\rightarrow K_2} [\![\mathrm{Fun}\,(X{:}K_1)\,T]\!]_\eta$.

S-OApp — By induction hypothesis, using the syntax-directedness of kinding.

S-Meet-G — By the induction hypothesis and Lemma 4.2.1(3).

S-Meet-LB — By Lemma 4.2.1(4).

S-Meet-Arrow — By Lemma 4.2.1(5).

S-Meet-All — By Lemma 4.2.1(6).

S-Meet-OAbs — By Lemma 4.2.1(7) (using syntax-directedness of kinding and Lemma 4.3.3).

S-Meet-OApp — By Lemma 4.2.1(8).

Note that, in the four last cases it is even true that $[\![S]\!]_\eta = [\![T]\!]_\eta$.

The *type context* $\Gamma/TV$ *obtained from a context* $\Gamma$ is defined in the obvious way:

$$\begin{aligned} \emptyset/TV &= \emptyset, \\ (\Gamma,\ x{:}T)/TV &= \Gamma/TV, \\ (\Gamma,\ X{\leq}T{:}K)/TV &= \Gamma/TV,\ X{\leq}T{:}K. \end{aligned}$$

**Theorem 4.3.5** *Let $\eta_1 \models \Gamma$ and $\eta_2 \models \Gamma$, such that $\eta_1/TV = \eta_2/TV$ and, for all $x \in dom(\Gamma)$, $\eta_1(x)\ \{[\![\Gamma(x)]\!]_\eta\}\ \eta_2(x)$, where $\eta = \eta_1/TV$. Then $[\![e]\!]_{\eta_1}\ \{[\![T]\!]_\eta\}\ [\![e]\!]_{\eta_2}$.*

**Proof:** From $\Gamma \vdash e \in T$ it follows by the well-kindedness of typing (3.3.12) that $\Gamma \vdash T \in \star$. Note that $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\Gamma/TV)$; then, by strengthening (3.3.5), $\Gamma/TV \vdash T \in \star$. Note also that $\eta \models \Gamma/TV$; by Lemma 4.3.1, $[\![T]\!]_\eta$ is defined and in $[\![\star]\!]$. We often use this fact implicitly in the following. The proof now proceeds by induction on a derivation of $\Gamma \vdash e \in T$.

T-Var — From the assumption that for every $x$ in $\mathrm{dom}(\Gamma)$. $\eta_1(x)\ \{[\![\Gamma(x)]\!]_\eta\}\ \eta_2(x)$.

T-ABS We are given that $\Gamma, x{:}T_1 \vdash e \in T_2$. Suppose that $p \in D$ and $q \in D$ are such that $p\ \{\llbracket T_1\rrbracket_\eta\}\ q$. Then $\eta_1[x{\leftarrow}p] \models \Gamma, x{:}T_1$ and $\eta_2[x{\leftarrow}q] \models \Gamma, x{:}T_1$. By the induction hypothesis, $\llbracket e\rrbracket_{\eta_1[x\leftarrow p]}\ \{\llbracket T_2\rrbracket_\eta\}\ \llbracket e\rrbracket_{\eta_2[x\leftarrow q]}$, that is, $\llbracket|\mathrm{erase}(e)|\rrbracket_{\eta_1[x\leftarrow p]}\ \{\llbracket T_2\rrbracket_\eta\}\ \llbracket|\mathrm{erase}(e)|\rrbracket_{\eta_2[x\leftarrow q]}$. From Lemma 4.1.1(2) it follows that $\llbracket\mathrm{fun}^\star(x)\,|\mathrm{erase}(e)|\rrbracket_{\eta_1}.\,p\ \{\llbracket T_2\rrbracket_\eta\}\ \llbracket\mathrm{fun}^\star(x)\,|\mathrm{erase}(e)|\rrbracket_{\eta_2}.\,q$, that is, $\llbracket\mathrm{fun}\,(x{:}T_1)\,e)\rrbracket_{\eta_1}.\,p\ \{\llbracket T_2\rrbracket_\eta\}\ \llbracket\mathrm{fun}\,(x{:}T_1)\,e\rrbracket_{\eta_2}.\,q$. Since $p$ and $q$ were chosen freely, we have $\llbracket\mathrm{fun}\,(x{:}T_1)\,e\rrbracket_{\eta_1}\ \{\llbracket T_1\rrbracket_\eta{\to}\llbracket T_2\rrbracket_\eta\}\ \llbracket\mathrm{fun}\,(x{:}T_1)\,e\rrbracket_{\eta_2}$, in other words $\llbracket\mathrm{fun}\,(x{:}T_1)\,e\rrbracket_{\eta_1}\ \{\llbracket T_1{\to}T_2\rrbracket_\eta\}\ \llbracket\mathrm{fun}\,(x{:}T_1)\,e\rrbracket_{\eta_2}$.

T-APP By the induction hypothesis, using the fact that $\llbracket f a\rrbracket_\eta = \llbracket f\rrbracket_\eta \llbracket a\rrbracket_\eta$.

T-TABS We are given that $\Gamma, X{\leq}T_1{:}K_1 \vdash e \in T_2$. Suppose that $P \subseteq^{K_1} \llbracket T_1\rrbracket_\eta$. Then $\eta_1[X{\leftarrow}P] \models \Gamma, X{\leq}T_1{:}K_1$ and $\eta_2[X{\leftarrow}P] \models \Gamma, X{\leq}T_1{:}K_1$. Since $\llbracket e\rrbracket_{\eta_1} = \llbracket e\rrbracket_{\eta_1[X\leftarrow P]}$, we have by the induction hypothesis that $\llbracket e\rrbracket_{\eta_1}\ \{\llbracket T_2\rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket e\rrbracket_{\eta_2}$. Since $P$ was chosen freely, we have that $\llbracket e\rrbracket_{\eta_1}\ \{\bigcap^\star_{P\subseteq^{K_1}\llbracket T_1\rrbracket_\eta} \llbracket T_2\rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket e\rrbracket_{\eta_2}$. Now the result follows from the definition $\llbracket - \rrbracket_\eta$ and the fact that $\llbracket\mathrm{fun}\,(X{\leq}T_1{:}K_1)\,e\rrbracket_\eta = \llbracket e\rrbracket_\eta = \llbracket e\rrbracket_{\eta[X\leftarrow P]}$.

T-TAPP We are given that $\Gamma \vdash f \in \mathrm{All}\,(X{\leq}T_1{:}K_1)\,T_2$ and that $\Gamma \vdash S \leq T_1$. By the induction hypothesis, $\llbracket f\rrbracket_{\eta_1}\ \{\llbracket\mathrm{All}\,(X{\leq}T_1{:}K_1)\,\rrbracket_\eta\}\ \llbracket f\rrbracket_{\eta_2}$, that is, $\llbracket f\rrbracket_{\eta_1}\ \{\bigcap^\star_{P\subseteq^{K_1}} \llbracket T_2\rrbracket_{\eta[X\leftarrow P]}\}\ \llbracket f\rrbracket_{\eta_2}$. By the well-kindedness of typing, syntax directedness of kinds, and well-kindedness of subtyping, $\Gamma \vdash S \in K_1$, and, by soundness of subtyping, $\llbracket S\rrbracket_\eta \subseteq^{K_1} \llbracket T_1\rrbracket_\eta$. So we have $\llbracket f\rrbracket_{\eta_1}\ \{\llbracket T_2\rrbracket_{\eta[X\leftarrow\llbracket S\rrbracket_\eta]}\}\ \llbracket f\rrbracket_{\eta_2}$, which, by Lemma 4.3.2 and the fact that $\llbracket f\rrbracket_\eta = \llbracket f S\rrbracket_\eta$ for any $\eta$, is $\llbracket f S\rrbracket_{\eta_1}\ \{\llbracket T_2[X{\leftarrow}S]\rrbracket_\eta\}\ \llbracket f S\rrbracket_{\eta_2}$.

T-FOR Immediate from the induction hypothesis, since $\mathrm{erase}(e[X{\leftarrow}S]) = \mathrm{erase}(e)$.

T-MEET We are given that $\Gamma \vdash \mathrm{ok}$ and also $\Gamma \vdash e \in T_i$ for each $i$ in $\{1\,..\,n\}$. The result follows by the induction hypothesis and the definitions of $\bigcap^\star$ and $\llbracket - \rrbracket_\eta$.

T-SUBSUMPTION We are given that $\Gamma \vdash e \in S$ and that $\Gamma \vdash S \leq T$. By the induction hypothesis, $\llbracket e\rrbracket_{\eta_1}\ \{\llbracket S\rrbracket_\eta\}\ \llbracket e\rrbracket_{\eta_2}$. By well-kindedness of typing $\Gamma \vdash S \in \star$, and by the soundness of subtyping $\llbracket S\rrbracket_\eta \subseteq^\star \llbracket T\rrbracket_\eta$. Hence, $\llbracket e\rrbracket_{\eta_1}\ \{\llbracket T\rrbracket_\eta\}\ \llbracket e\rrbracket_{\eta_2}$.

**Corollary 4.3.6 (Soundness of typing)** *Let $\eta$ be an environment such that $\eta \models \Gamma$. Then $\Gamma \vdash e \in T$ implies $\llbracket e\rrbracket_\eta {\in} dom(\llbracket T\rrbracket_\eta)$.*

**Proof:** Take $\eta_1 = \eta_2 = \eta$.

# 5 Encoding Multiple Inheritance

We close with a full implementation of the `extend2` function, generalizing the `extend` function in Section 7 of [PT93]. As we suggested in the introduction, an intersection type must be used at one point (marked `***` in the definition of `extend`) to obtain a sound typing; the rest is straightforward.

This implementation of classes and inheritance makes the local state of each class inaccessible both to clients of objects and to methods defined in subclasses. Other variations are possible; we chose this one to simplify the presentation of Section 2.

If `M` is an object interface — an operator of kind `*->*` — then `Class(M)` is the set of classes whose instances have type `Object(M)`. Each such class consists of a local representation type `MyR` (whose identity is hidden by an existential quantifier), an element `initstate`∈`MyR` that is used as the initial value of the state in new objects created from this class, and a function `buildM` that can be used to construct the methods of the new objects:

```
# Class =
#   Fun(M:*->*)
#     Some(R)
#       {| initstate: R,
#          buildM: ClassMethods M R |};
Class : (*->*)->*
```

To cope with different representations of local state in subclasses, the method-building function is abstracted on two parameters: a type `FinalR` representing the "full" state of an eventual subclass, and an "extractor" giving access to the components of interest to the methods being built. The method builder is also abstracted on a collection of `self`-methods of the same types as its own methods. Given these, it yields a concrete collection of methods specialized to work properly in an object with representation type `FinalR`:

```
# ClassMethods =
#   Fun(MyM:*->*)
#   Fun(MyR)
#     All(FinalR)
#     (Extractor FinalR MyR) ->
#     (MyM(FinalR)) ->
#     (MyM(FinalR));
ClassMethods : (*->*)->*->*
```

Finally, an extractor is just a pair of maps, `get` and `put`.

```
# Extractor = Fun(SS) Fun(TT) {| get: SS->TT, put: SS->TT->SS |};
Extractor : *->*->*
```

Intuitively, `get` extracts the "superclass part" of an element of a subclass's state, while `put` overwrites the superclass part, yielding a new subclass state.

For example, the point class of Section 2 can be defined directly (rather than as an extension of `thingClass`) as follows:

```
# pointClass =
#   < {|x:Int|},
#     {initstate = {x=0},
#      buildM = fun(FinalR)
#               fun(e: Extractor FinalR {|x:Int|})
#               fun(self: PointM FinalR)
#                 {getX = fun(s:FinalR) (e.get s).x,
#                  setX = fun(s:FinalR) fun(i:Int) e.put s {x=i},
#                  bump = fun(s:FinalR) e.put s {x = plus 1 (e.get s).x}
#                 }}
#    > : Class PointM;
pointClass = <val> : Class PointM
```

(We use the ascii syntax "`<R,b>:T`" for introducing elements of existential types: `R` is the hidden witness type, `b` is the body, and `T` is the existential type where the result is to live. The corresponding elimination form is written "`open e as <R,x> in b`.")

A class with two superclasses generates objects whose internal states have three parts: one for each superclass and one for the new components local to the class itself. For example, an instance of `cpointClass` contains a point state of type `{|x:Int|}`, a colored-object state of type `{|c:Color}}`, and an empty local state. The `extend2` function takes two classes, an initial local state, and a function for incrementally building a collection of new methods from the old ones, and constructs a subclass of this form.

```
# extend2 =                                          %Arguments:
#  fun(SuperM1: *->*)                                   %first superclass interface
#  fun(SuperM2: *->*)                                   %second superclass interface
#  fun(MyM < SuperM1/\SuperM2)          %***     %new subclass interface
#  fun(MyLocalR: *)                                     %local state type
#  fun(superClass1: Class SuperM1)                      %first superclass
#  fun(superClass2: Class SuperM2)                      %second superclass
#  fun(myinitstate: MyLocalR)                           %initial local state
#  fun(mymethods:                                       %incr method extension fun with args
#         All(FinalR)                                     %a final representation type
#           (Extractor FinalR MyLocalR) ->                %an extractor for the local state
#           (SuperM1(FinalR)) ->                          %the first superclass's methods
#           (SuperM2(FinalR)) ->                          %the second superclass's methods
#           (MyM(FinalR)) ->                              %the self-methods
#           (MyM(FinalR)))                                %returning the new methods
#  open superClass1 as <SuperR1,superData1> in   %open the first superclass
#  open superClass2 as <SuperR2,superData2> in   %open the second superclass
#  let MyR = Triple SuperR1 SuperR2 MyLocalR in  %define the new state type (a triple)
#   <                                          %Result: a new class (an existential)
#    MyR,                                               %with state type MyR
#    {initstate = triple SuperR1 SuperR2 MyLocalR%and initial state a triple of
#                    (superData1.initstate)       %first super's initial state
#                    (superData2.initstate)       %second super's initial state
#                    myinitstate,                 %local initial state
```

```
#     buildM =                                    %and method-builder, a fun with args
#      fun(FinalR)                                  %a final representation type
#      fun(e: Extractor FinalR MyR)                 %and an extractor for the state
#      fun(self: MyM(FinalR))                       %and a collection of self-methods...
#        let eself =
#          composeExtractors
#             FinalR MyR MyLocalR e
#             (extract3of3 SuperR1 SuperR2 MyLocalR) in
#        let esuper1 =
#          composeExtractors
#             FinalR MyR SuperR1 e
#             (extract1of3 SuperR1 SuperR2 MyLocalR) in
#        let esuper2 =
#          composeExtractors
#             FinalR MyR SuperR2 e
#             (extract2of3 SuperR1 SuperR2 MyLocalR) in
#        mymethods FinalR eself                   %returning methods built by mymethods
#          (superData1.buildM FinalR esuper1 self)   %when applied to concrete methods
#          (superData2.buildM FinalR esuper2 self)   %of the two superclasses
#          self                                      %and the self-methods
#        end end end}
#     > : Class MyM
#   end end end;
extend2 = <val>
        : All(SuperM1)
          All(SuperM2)
          All(MyM<SuperM1/\SuperM2)
          All(MyLocalR)
               (Class SuperM1)
            -> (Class SuperM2)
            -> MyLocalR
            -> (All(FinalR)
                    (Extractor FinalR MyLocalR)
                  ->(SuperM1 FinalR)
                  ->(SuperM2 FinalR)
                  ->(MyM FinalR)
                  ->(MyM FinalR))
            -> (Class MyM)
```

This definition uses a utility function for composing extractors in the obvious way:

```
# composeExtractors =
#   fun(T1) fun(T2) fun(T3)
#     fun(e1: Extractor T1 T2)
#       fun(e2: Extractor T2 T3)
#         {get = fun(t1:T1) e2.get (e1.get t1),
#          put = fun(t1:T1) fun(t3:T3)
#                  e1.put t1 (e2.put (e1.get t1) t3)};
```

```
composeExtractors = <val>
                  : All(T1)
                    All(T2)
                    All(T3)
                         (Extractor T1 T2)
                      -> (Extractor T2 T3)
                      -> {|get:T1->T3, put:T1->T3->T1|}
```

For forming triples, we use the type abbreviation

```
# Triple = Fun(T1) Fun(T2) Fun(T3) {| fst:T1, snd:T2, thd:T3 |};
Triple : *->*->*->*
```

with the constructor

```
# triple =
#   fun(T1) fun(T2) fun(T3)
#   fun(t1:T1) fun(t2:T2) fun(t3:T3)
#     {fst=t1, snd=t2, thd=t3};
triple = <val>
       : All(T1)
         All(T2)
         All(T3)
           T1 -> T2 -> T3 -> {|fst:T1, snd:T2, thd:T3|}
```

and the projections

```
# extract1of3 =
#  fun(T1) fun(T2) fun(T3)
#    {get = fun(p: Triple T1 T2 T3) p.fst,
#     put = fun(p: Triple T1 T2 T3) fun(t:T1) {fst=t, snd=p.snd, thd=p.thd} };
extract1of3 = <val>
            : All(T1)
              All(T2)
              All(T3)
                {|get: (Triple T1 T2 T3)->T1,
                  put: (Triple T1 T2 T3)->T1->{|fst:T1, snd:T2, thd:T3|}|}
```

and `extract2of3` and `extract3of3`, which are defined similarly.

# 6 Discussion

For technical convenience, our model of multiple inheritance has been presented in a purely functional setting: sending a message that changes the internal state of an object yields a new object with appropriately modified state rather than updating the original object's state in-place. Extending the model to an object model including mutable internal state should be fairly straightforward, following [Pie93b]. The principal difficulty lies in the fact that the type $\top^\star$, which classifies *all* terms (not only the well-behaved ones), is only sensible in a call-by-name setting, where we can be sure that an ill-behaved subterm will never be needed during evaluation. To move to a call-by-value setting, where the order of updates to mutable variables is easier to predict, we need to replace $\top^\star$ by $F_{\leq}$'s less permissive maximal type *Top* and restrict the formation rules so that only nonempty intersections are allowed.

We have seen that a simple semantics of intersections can be given by a straightforward extension of a model of higher-order polymorphism. Indeed, it appears that even the combination of intersections and recursive types is semantically unproblematic [Cardone, personal communication, 1992]. This suggests that other existing accounts of typed object-oriented programming could be extended with multiple inheritance by a construction similar to ours. Indeed, accounts such as that of Bruce [Bru93], which interpret a high-level object syntax directly into a denotational model, should support this extension almost trivially: the need for intersection *types* (as opposed to intersections in the model) arose here from our requirement that the `extend` function itself be a term of the target $\lambda$-calculus. Extending accounts based on a translation into a $\lambda$-calculus with recursive record types, on the other hand, may be more difficult, since no syntactic study has been made so far of a type system combining intersections and recursive types.

## Acknowledgements

We are grateful for discussions with Henk Barendregt, Kim Bruce, Luca Cardelli, Bob Harper, John Reynolds, David Turner, and Benjamin Werner.

This work was supported by funding from the U.K. Science and Engineering Research Council and the ESPRIT Basic Research Actions TYPES and CONFER, and from the Netherlands Organization for Scientific Research (N.W.O.).

# References

[Aba93] Martín Abadi. Baby Modula-3 and a theory of objects. Research Report 95, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1993.

[AC93] Martín Abadi and Luca Cardelli. Object calculi with override. Draft report, April 1993.

[Bar84] H. P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. To appear in [GM93]. An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[BM92] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[BMM90] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. In Huet [Hue90], pages 213–272. Also appeared in Information and Computation 84, 1 (January 1990).

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished notes, October 1990.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. To appear in [GM93].

[Cas92] Giuseppe Castagna. Strong typing in object-oriented paradigms. Rapport de Recherche LIENS-92-11, Ecole Normale Supérieure, Paris, May 1992.

[CC90] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 19–76. Academic Press, 1990.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CDC78] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. To appear in [GM93].

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM conference on LISP and Functional Programming*, pages 182–192, San Francisco, July 1992. ACM Press. Also available as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. To appear in [GM93].

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CP94] Giuseppe Castagna and Benjamin Pierce. Decidable bounded quantification. In *Proceedings of Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, OR*. ACM, January 1994. To appear.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[FMRS90] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Fifth Annual Symposium on Logic in Computer Science (Philadelphia, PA)*, pages 346–354. IEEE Computer Society Press, June 1990.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GM93] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1993. To appear.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HP92] Martin Hofmann and Benjamin Pierce. An abstract view of objects and subtyping (preliminary report). Technical Report ECS-LFCS-92-226, University of Edinburgh, LFCS, 1992.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and $\lambda$-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[Hue90] Gérard Huet, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993. To appear.

[Mit90a] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. To appear in [GM93].

[Mit90b] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In Huet [Hue90], pages 195–212.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[Pie93a] Benjamin C. Pierce. A model of delegation based on existential types. Available electronically, April 1993.

[Pie93b] Benjamin C. Pierce. Mutable objects. Available electronically, June 1993.

[PT93] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

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

[Sco76] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Symposium on Object-Oriented Programming: Languages, Systems, and Applications (OOPSLA)*, pages 227–241, 1987.