Foundations for Virtual Types

Atsushi Igarashi   Benjamin C. Pierce
Department of Computer & Information Science
University of Pennsylvania
200 South 33rd St.
Philadelphia, PA 19104, USA
{igarasha,bcpierce}@saul.cis.upenn.edu

October 2, 1998

Abstract

Virtual types have been proposed as a notation for generic programming in object-oriented languages, as an alternative to the more familiar mechanism of *parametric classes*. The tradeoffs between the two mechanisms are a matter of current debate: for many examples, both appear to offer convenient (indeed almost interchangeable) solutions; in other situations, one or the other seems to be more satisfactory. However, it has proved difficult to draw rigorous comparisons between the two approaches, partly because current proposals for virtual types vary considerably in their details, and partly because the proposals themselves have been described rather informally, often in the complicating context of full-scale language designs.

Work on the foundations of object-oriented languages has already established a clear connection between parametric classes and the polymorphic functions found in familiar typed lambda-calculi. Our aim here is to explore a similar connection between virtual types and dependent records.

We present, by means of examples, a straightforward model of objects with embedded type fields in a typed lambda-calculus with subtyping, type operators, fixed points, and dependent records with both "bounded" and "manifest" type fields. Using this model, we then discuss and compare some features between previous proposals and show why some can be checked statically while others require run-time checks.

1 Introduction

Language support for generic programming plays an important role in the development of reusable libraries. In the domain of object-oriented languages, two approaches to genericity have been considered. The more familiar—based closely on the classical *parametric polymorphism* of functional languages such as ML and Haskell—can be found, for example, in the *template* mechanism of C++ [Str97] and the *parametric classes* of a number of proposed extensions to Java [OW97, MBL97, AFM97, etc.]. An alternative approach, commonly called *virtual types* (or *virtual classes*), allows classes and objects to contain types as members, in addition to the usual field and method members.[1] Virtual types were originally developed in BETA [MMP89] and have recently been proposed for Java [Tho97].

The static typing of virtual types is not yet clearly understood. Indeed, early proposals were statically unsafe, requiring extra runtime checks; more recent work has produced several proposals for type-safe variants [Tor98, BOW98]. These proposals vary substantially in their details, and have generally been presented in rather informal terms (and in the complicating context of full-scale language designs), making them difficult to evaluate and compare.

[1] Referring to this approach with the phrase "virtual types" may somewhat confusing, since—as we will see—these type members may or may not be "virtual" in the sense of *virtual* or *abstract methods*. But the terminology is standard.

Our primary goal in this paper is to establish a rigorous setting in which to understand and discuss the basic mechanisms of virtual types. Following a long line of past work on foundations for object-oriented programming (see [BCP97] for history and citations), we model objects and classes with virtual types as a particular style of programming in a fairly standard typed lambda-calculus. On this basis, we examine (1) the power of the type-theoretic features that seem to be required for modeling virtual types, (2) the similarities and differences between existing proposals, and [in the full version] (3) the type-theoretic intuitions behind the much-discussed "overlap" between virtual types and parametric classes in practice.

The paper is organized as follows. Section 2 reviews the idea of virtual types by means of a standard example—the Animal/Cow class hierarchy of Shang [Sha96]. Section 3 sketches the main features of the typed lambda-calculus that forms the setting for our model. (The calculus is defined in full in Appendix A.) Section 4 develops the Animal/Cow example in this calculus. Section 5 summarizes some previous work on virtual types in the light of our lambda-calculus model. Section 6 adds a few comments on several smaller issues. For the sake of brevity, we concentrate on the earlier sections in this summary, emphasizing the definition of our typed lambda-calculus and development of the basic Animal/Cow example and just sketching the discussion and comparisons in later sections.

2 Virtual Types

We begin by reviewing the notion of virtual types through an example. This example, used throughout the paper, is a variant of the "Animal/Cow" example by David Shang [Sha96]. Our notation is Java-like, but does not exactly correspond to any of the existing proposals for virtual types in Java.

We begin by defining a generic class of animals, along with its interface.

```
interface AnimalI {
  type FoodType <: Food;
  void eat (FoodType f);
  void eatALot (FoodType f); }
```

```
virtual class Animal implements AnimalI {
  virtual type FoodType <: Food;
  virtual void eat (FoodType f);
  void eatALot (FoodType f) {
    eat(f);
    eat(f); }}
```

Every animal has methods `eat` and `eatALot`, both accepting some food as an argument. The body of the `eat` method, which is specific to particular kinds of animals, is omitted; the `virtual` marker defers the responsibility of providing an implementation to subclasses. (We use the C++ keyword `virtual` in preference to Java's `abstract` to avoid terminological confusion: locutions like "abstract type" already have a well-established meaning.) The calls to `eat` from the body of the `eatALot` method will call whatever body is provided by the subclass.

Similarly, the class `Animal` defers specifying exactly what kind of food a given kind of animal likes to eat. The virtual member `FoodType` acts as placeholder for this type, allowing it to be mentioned in the types of `eat` and `eatALot`, just as the declaration of `eat` provides a placeholder for its eventual implementation, allowing it to be referred to from the body of `eatALot`. Classes with virtual members (either types or methods) cannot be instantiated, since they are incomplete: they can only be subclassed.

The interface `AnimalI` specifies that every animal object has three members: a type `FoodType` and methods `eat` and `eatALot`. The `FoodType` member of every animal is known to be some kind of `Food` (`FoodType<:Food`), but, since different animals eat different kinds of food, the exact identity of this type is not visible. It follows immediately that it is not possible to feed an animal without knowing what kind of animal it is: if `a` is an object of type `AnimalI`, then `a`'s `eat` method requires an argument of type `a.FoodType`; but there is no way to obtain a value of this type (except, perhaps, by building a nutrient-free empty value using `new`).

Specific kinds of animals are modeled by classes inheriting from `Animal`. For example, here is a `Cow` class and its interface:

```
interface CowI extends AnimalI {
  type FoodType ↔ Grass; }
```

```
class Cow extends Animal implements CowI {
  final type FoodType ↔ Grass;
  void eat (FoodType f) { ... }}
```

In Cow, the virtual method eat is given a concrete implementation (shown as "..."). Similarly, the virtual type member FoodType is given a concrete value, Grass. The annotation final on the FoodType member means that it cannot be redefined by subclasses: every subclass of Cow is guaranteed to have Grass as its FoodType. The interface CowI reflects the fact that FoodType is final: in effect, it tells the world that every cow eats food whose type is *equal* to Grass. Thus, given an object a of type CowI, we may validly obtain some grass from any source and pass it to the eat or eatALot methods.

Virtual types are also useful in more standard examples of generic programming. For example, a generic Set class can be defined with a virtual type ElementType. Then classes IntegerSet, StringSet, etc. can be defined by inheriting from Set and giving ElementType a final binding to Integer or String. Other examples of generic programming with virtual types can be found in [MMP89, Tho97].

3 Records with Type Fields

It is well known that parametric classes—classes abstracted on type parameters—can be understood as polymorphic functions in a typed lambda-calculus. Similarly, objects with type members should clearly be modeled as some kind of records with type fields. Fortunately, such records have been studied extensively in the type-theory literature (e.g. [Car91]). Indeed, even the particular form of the constraints on type members appearing in the interfaces AnimalI (FoodType<:Food) and CowI (FoodType$\leftrightarrow$Grass) correspond to well-known constructions. Records with type fields constrained by <: are a generalization of *partially abstract types* [CW85]; records with type fields constrained by $\leftrightarrow$ correspond to *translucent* or *manifest* sums in the typed lambda-calculi used by Harper and Lillibridge [HL94, Lil97] and Leroy [Ler94] to model module systems.

The lambda-calculus sketched in this section is based directly on these intuitions. In essence, it can be described as System $F^{\omega}_{\leq}$ (the omega-order polymorphic lambda-calculus with subtyping [Car90, CL91, PS94, Com94]) plus dependent records with both "bounded" and "manifest" type fields. For brevity, we assume readers are already familiar with System $F^{\omega}_{\leq}$ and concentrate on explaining dependent records, which are somewhat less familiar; Appendix A includes a summary of the whole system.

To support records with type fields, some fairly sophisticated machinery is required. First, we must deal with the fact that later fields in a record may refer to earlier fields by name—e.g., the type of the eat field must refer to the FoodType field. (Thus, in particular, the order of fields is significant.) Second, we must be able to deal with record-projection expressions like a.FoodType appearing in the types of values like a.eat.

In general, a dependent record has the form $\{\mathtt{l}_i \triangleright \exists \beta_i{}^{i\in 1\cdots n}\}$, where each β_i is a *field* of one of two forms: either a term field $\mathtt{x}_i\mathtt{=}\mathtt{t}_i$ or a type field $\mathtt{X}_i\mathtt{=}\mathtt{T}_i$. The label $\mathtt{l}_i$ is the *external name* of the i-th field—the name that is used to project it from outside. The *internal name* $\mathtt{x}_i$ or $\mathtt{X}_i$ is a binder whose scope is the rest of the fields in the record; it has no significance outside of the record definition and can be renamed at will according to the usual conventions of alpha-conversion. (The distinction between internal and external names is discussed, for example, in [HL94].) When the internal and external names agree, we write $\exists \triangleright \mathtt{x}_i\mathtt{=}\mathtt{t}_i$ as a shorthand for $\mathtt{x}_i \triangleright \exists \mathtt{x}_i\mathtt{=}\mathtt{t}_i$, and similarly $\exists \triangleright \mathtt{X}_i\mathtt{=}\mathtt{T}_i$ for $\mathtt{X}_i \triangleright \exists \mathtt{X}_i\mathtt{=}\mathtt{T}_i$. When the internal name x is not mentioned in the rest of the record, we drop it, writing simply $\mathtt{l}\triangleright\mathtt{t}$ instead of $\mathtt{l}\triangleright\exists\mathtt{x}\mathtt{=}\mathtt{t}$. For example, the record value $\mathtt{r = \{X\triangleright\exists X=Nat,l\triangleright\exists x=}\lambda\mathtt{[y:X]y+1\}}$ is abbreviated $\mathtt{\{\exists\triangleright X=Nat,l\triangleright}\lambda\mathtt{[y:X]y+1\}}$.

A record type has the form $\{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\}$, where each $\mathtt{B}_i$ is a *binding* of one of three forms: a *term binding* $\exists\mathtt{x:T}$, a *bounded type binding* $\exists\mathtt{X<:T}$, or a *manifest type binding* $\exists\mathtt{X}\leftrightarrow\mathtt{T}$. For example, the record r above has type $\{\!|\, \mathtt{X\triangleright\exists X}\leftrightarrow\mathtt{Nat,l\triangleright\exists x:X}\rightarrow\mathtt{X} \,|\!\}$, which we abbreviate $\{\!|\, \mathtt{\exists\triangleright X}\leftrightarrow\mathtt{Nat,l\triangleright X}\rightarrow\mathtt{X} \,|\!\}$. A less informative type also possessed by r is $\{\!|\, \mathtt{\exists\triangleright X<:Top,l\triangleright X}\rightarrow\mathtt{X} \,|\!\}$, which hides the representation of X (and corresponds to the usual existential type $\exists\mathtt{X.X}\rightarrow\mathtt{X}$).

Formally, the typing rule for record introduction is:

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \beta_j : \mathtt{B}_j{}^{\,j\in 1\cdots n} \qquad \Gamma \vdash \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\} : *}{\Gamma \vdash \{\mathtt{l}_i \triangleright \exists \beta_i{}^{i\in 1\cdots n}\} : \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\}} \qquad \text{(T-RCD)}$$

Each field definition β_i must satisfy the corresponding binding $\mathtt{B}_i$ under a context augmented with the information of the preceding fields $(\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{i-1})$. Term fields $\mathtt{x}_i\mathtt{=}\mathtt{t}_i$ satisfy bindings of the form $\mathtt{x}_i\mathtt{:}\mathtt{T}_i$; type fields $\mathtt{X}_i\mathtt{=}\mathtt{T}_i$ satisfy manifest type bindings $\mathtt{X}_i\leftrightarrow\mathtt{T}_i$. (Note that we cannot directly derive a record type with a bounded type binding using the rule T-RCD. For example, the type given to `r` above is $\{\!|\, \exists \triangleright \mathtt{X} \leftrightarrow \mathtt{Nat}, \mathtt{l}{:}\mathtt{X} \to \mathtt{X} \,|\!\}$. If we want to hide the identity of `X` and give `r` the abstract type $\{\!|\, \exists \triangleright \mathtt{X} <: \mathtt{Top}, \mathtt{l}{:}\mathtt{X} \to \mathtt{X} \,|\!\}$, we use the usual subsumption rule plus the record subtyping rules discussed below.)

The rule for record projections is basically the same as the standard record elimination rule: if a field `l` of `t` has binding $\exists \mathtt{x}{:}\mathtt{T}$, then `t.l` has type `T`. If `T` depends on other fields—that is, if the internal name $\mathtt{X}_i$ of a preceding field $\mathtt{l}_i$ occurs free in `T`—then the corresponding record projection $\mathtt{t.l}_i$ should be substituted for $\mathtt{X}_i$ to prevent the internal name from escaping its scope.

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} \qquad \mathtt{B}_j{=}\mathtt{x}{:}\mathtt{T}}{\Gamma \vdash \mathtt{t.l}_j : \{BV(\mathtt{B}_1) \mapsto \mathtt{t.l}_1\} \cdots \{BV(\mathtt{B}_{j-1}) \mapsto \mathtt{t.l}_{j-1}\}\mathtt{T}} \quad \text{(T-DOT)}$$

We write $BV(\mathtt{B})$ for the bound variable of the binding `B`; that is, $BV(\mathtt{x}{:}\mathtt{T}) = \mathtt{x}$, $BV(\mathtt{X} \leftrightarrow \mathtt{T}) = \mathtt{X}$, and $BV(\mathtt{X} <: \mathtt{T}) = \mathtt{X}$.

The subtyping rule for record types is:

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{B}_j <: \mathtt{B'}_j{}^{j \in 1 \cdots n} \qquad \vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{n+k} \;\; \mathtt{ok} \qquad \vdash \Gamma, \mathtt{B'}_1, \ldots, \mathtt{B'}_n \;\; \mathtt{ok}}{\Gamma \vdash \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i \in 1 \cdots n+k} \,|\!\} <: \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B'}_i{}^{i \in 1 \cdots n} \,|\!\}} \quad \text{(S-RCD)}$$

As usual for ordinary (non-dependent) records, "width subtyping" is allowed: extra fields (the n+1-st to n+k-th fields) can be dropped. Also, corresponding bindings $\mathtt{B}_i$ and $\mathtt{B'}_i$ are compared using a *sub-binding* relation. When both are term bindings—i.e., $\mathtt{B}_i$ and $\mathtt{B'}_i$ are of the form `x:S` and `x:T`—`S` should be a subtype of `T`: this captures ordinary "depth subtyping." For type bindings, we have $(\mathtt{X} \leftrightarrow \mathtt{T}) <: (\mathtt{X} <: \mathtt{S}) <: (\mathtt{X} <: \mathtt{U})$ if $\mathtt{T} <: \mathtt{S} <: \mathtt{U}$; the first clause ($(\mathtt{X} \leftrightarrow \mathtt{T}) <: (\mathtt{X} <: \mathtt{S})$) allows the exact identity of a type field to be replaced with an upper bound; the second ($(\mathtt{X} <: \mathtt{S}) <: (\mathtt{X} <: \mathtt{U})$), corresponding to subtyping of bounded existential types, allows to loosen the bound of `X`. For example, we can derive $\{\!|\, \exists \triangleright \mathtt{X} \leftrightarrow \mathtt{Nat}, \mathtt{l} \triangleright \mathtt{X} \to \mathtt{X} \,|\!\} <: \{\!|\, \exists \triangleright \mathtt{X} <: \mathtt{Top}, \mathtt{l} \triangleright \mathtt{X} \to \mathtt{X} \,|\!\}$. (As usual, this rule leads to an undecidable subtyping relation [Pie94, Lil97].)

4 Encoding Virtual Types

With this overview of our typed lambda-calculus now in hand, we can proceed to the technical heart of the paper: a straightforward encoding of the animal example from Section 2 in terms of records with type fields. For the sake of concreteness, we extend the familiar *existential encoding* of objects [PT94, HP95] (we believe that other methods for encoding objects, using recursive types or both recursive and existential types [BCP97], would yield similar results, but have not tried in detail). For brevity, we assume that the reader is already familiar with the idea of the existential encoding.

To avoid introducing additional complexities in the type theory, we give an encoding of *purely functional* objects; for example, we assume that an animal's `eat` method returns a new animal rather than side-effecting the instance variables of the receiver. Imperative variants of the encoding are discussed in Section 6.

4.1 Interfaces

The interface of an object is represented as a type operator $\lambda[\mathtt{Rep}]\{\!|\, \mathtt{m}_i \triangleright \mathtt{T}_i{}^{i \in 1 \cdots n} \,|\!\}$, where the bound variable `Rep` stands for the type of the object's internal state and each $\mathtt{T}_i$ is the type of the corresponding method $\mathtt{m}_i$. Each method takes the internal state of the object as an explicit argument. Thus, the interface `AnimalI` is represented as:

```
AnimalI = λ[Rep] {| ∃▷FoodType<:Food, eat▷Rep→FoodType→Rep, eatALot▷Rep→FoodType→Rep |};

AnimalI : * → *
```

The binding `FoodType<:Food` is a direct transliteration of the constraint on `FoodType` in Section 2. Similarly, the interface `CowI` is represented as

```
CowI = λ[Rep] {| ∃▷FoodType↔Grass, eat▷Rep→FoodType→Rep, eatALot▷Rep→FoodType→Rep |};
```

```
CowI : * → *
```

where the binding of FoodType is now manifest. Note that CowI is a subtype of AnimalI; this will allow Cow objects to be regarded as animals.

4.2 Objects

The type of objects with interface I, written Object I, is a record type including a hidden (existentially bound) representation type Rep, a state field of type Rep, and a method vector field containing a record of type I Rep. Formally, the type operator Object is defined as:

```
Object = λ[I:*→*]{|∃▷Rep, state▷Rep, methods▷I Rep|};

Object : (*→*) → *
```

Choosing the simple representation

```
CowR = {|hungry▷Bool|};

CowR : *
```

for the internal state of cows, we can define an element of the type Object CowI as follows:

```
cow = {∃▷Rep=CowR,
       state▷ {hungry▷true},
       methods▷ fix (CowI CowR)
                  λ[self:CowI CowR]
                    {∃▷FoodType=Grass,
                     eat▷λ[s:CowR]λ[f:FoodType]{hungry▷false},
                     eatALot▷λ[s:CowR]λ[f:FoodType]
                                (self.eat (self.eat s f) f)}}
      :: Object CowI;

cow : Object CowI
```

4.3 Classes

In the full version of the paper, this example is extended to include a *class* of animals and a derived subclass of cows. This development follows familiar lines [PT94], but there are some interesting subtleties concerning the typing of methods like eat whose types involve virtual types. To give a flavor of this development, here—without explanation—is the encoding of the animal class:

```
AnimalR = {|hungry▷Bool|};
animalClass = λ[self:AnimalI AnimalR]
                {∃▷FoodType=self.FoodType,
                 eat▷self.eat,
                 eatALot▷λ[s:AnimalR]λ[f:FoodType] self.eat (self.eat s f) f}
            :: Π[self: AnimalI AnimalR]
                 {|∃▷ FoodType↔self.FoodType,
                   eat▷ AnimalR→FoodType→AnimalR,
                   eatALot▷ AnimalR→FoodType→AnimalR|};
```

5 Comparisons

Virtual types (called *virtual classes* in the original proposal) were first introduced in BETA [MMPN93] by Madsen and Møller-Pedersen [MMP89] as a mechanism to achieve genericity in object-oriented languages. Later, Thorup [Tho97] introduced virtual types as an extension for Java. In all of this work, virtual types in classes are in fact *not* actually virtual in our sense: the interface of animal objects, according to their view, would better be modeled by

```
AnimalI = λ[Rep] ⦃∃▷FoodType↔Food, eat▷Rep→FoodType→Rep, eatALot▷Rep→FoodType→Rep⦄;
```

where `FoodType` is declared equal to `Food`. However, they also allow type fields to be specialized, so that

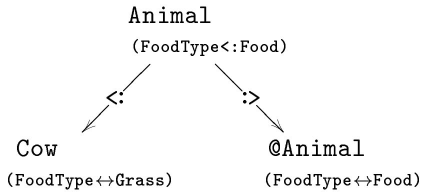
```
CowI = λ[Rep] ⦃∃▷FoodType↔Grass, eat▷Rep→FoodType→Rep, eatALot▷Rep→FoodType→Rep⦄;
```

as before. Finally, they want to regard cows to be animals, i.e., `CowI <: AnimalI` and `Object CowI <: Object AnimalI`. Taken together, these properties (specifically, the inclusion `CowI <: AnimalI`) yield an unsound type system: we can take a cow, regard it as an animal, and feed it some meat (which has type `Meat`, a subtype of `Food`, and hence an acceptable argument to an `Animal`'s `eat` method).[2]

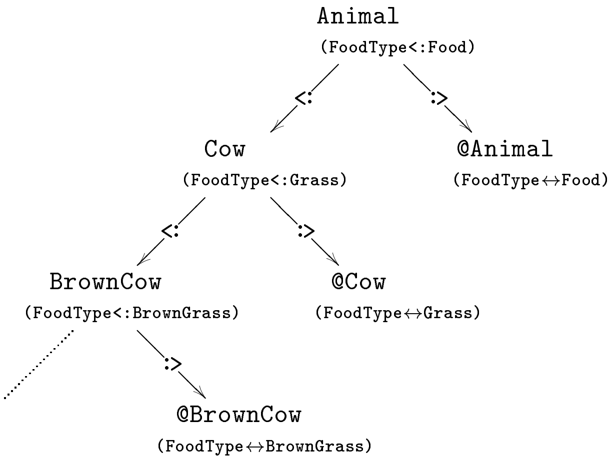
Various approaches have been suggested to remedy this unsoundness. In BETA and in Thorup's proposed Java extension, run-time checks are added to methods like `eat` to make sure that their arguments are actually acceptable. (In [MMMP90], it is observed that these checks can be omitted in the case where a type binding is marked `final`.)

Torgersen [Tor98] proposed a statically typesafe variant of virtual types, introducing essentially the same distinction as we have made between virtual type bindings (which may be specialized in subclasses, but which block instantiation of the classes containing them) and final ones (which allow instantiation but block further specialization in subclasses). Our model of objects with virtual types corresponds closely to his proposal.

A possible criticism of Torgersen's idea is that, in general, it may lead to duplication of the class hierarchy. For one thing, if the class `Animal` contains virtual types but no virtual methods (i.e., if `eat` is given a concrete generic implementation), then we may want to instantiate the class `Animal` itself. This requires making an explicit subclass (let's call it `@Animal`) of `Animal` in which `FoodType` is equal to `Food`.



Also, rather than making `Cow` a leaf of the subclass hierarchy, we may wish to allow further specialization in subclasses. In this case, we should change the constraint on `FoodType` to `<:Grass`, make `Cow` a virtual class, and introduce another leaf class `@Cow` in which `FoodType↔Grass`.



Fortunately, the `@` variants can be derived mechanically from the other classes, as Torgersen pointed out. More recently, Bruce, Odersky and Wadler [BOW98] have proposed another statically safe variant of virtual types, which can be viewed as making this idea explicit. (They do not present their proposal in this light, but we find this to be the most helpful way of understanding what they did.) In their system, virtual types are

[2] Large-scale experiments on British farms have shown that this practice leads to undesirable results.

always introduced with `<:` constraints (they write "`FoodType as Food`"); for each class `C`, the "exact" class `@C` is automatically provided. The `new` operator generates instance of exact classes, so that the expression `new Cow` yields an object of type `@Cow`, which can be regarded as a `Cow` by forgetting its "exactness," and further regarded as an `Animal` (but not an `@Animal`) by ordinary subtyping.

6 Discussion

We close by discussing some additional points related to our encoding. For brevity in this summary, we just sketch the topics treated in the full version.

- The "overlap" between virtual types and parametric classes has been remarked by several authors [BOW98, TT98]. In the encoding, it can be viewed as a consequence of the inter-encodability of universal and existential polymorphism in the presence of dependent types.
- There are several points where the modeling of objects and classes with virtual types becomes surprisingly difficult. We are currently struggling with...
  - imperative variants of the encodings, where methods like `eat` work by side-effecting mutable instance variables [the technicalities of the underlying type theory become astonishingly subtle when imperative features are added];
  - instance variables of virtual types; and
  - mutually recursive classes involving virtual types, such as the well-known subject-observer example.

  The first of these can be viewed just as a weakness of the modeling framework—a limitation of the current state of development of type systems combining dependent types and computational effects. The others, though, may turn out to be signs of unrecognized subtleties in the intended semantics of virtual types themselves.

Acknowledgments

This work was supported by Indiana University, the University of Pennsylvania, and the National Science Foundation under grant CCR-9701826, *Principled Foundations for Programming with Objects*. Igarashi is a research fellow of the Japan Society for the Promotion of Science.

References

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, October 1997.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

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

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[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.

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, Portland, OR, January 1994.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parametrized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 1997.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, pages 140–150, Ottawa, ON Canada, October 1990. ACM Press, New York, NY , USA. Published as SIGPLAN Notices, volume 25, number 10.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, 1989.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[PS94] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. 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".

[Sha96] David Shang. Are cows animals? *Object Currents 1*, 1996. `http://www.sigs.com/objectcurrents/`.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, third edition, 1997.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, 1997.

[Tor98] Mads Torgersen. Vitrual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[TT98] Kresten Krab Thorup and Mads Torgersen. Unifying genericity, 1998. Manuscript.

A Summary of Calculus

This appendix summarizes the syntax, reduction rules, and typing rules of a higher-order polymorphic λ-calculus with subtyping (System $F^{\omega}_{\leq}$) plus dependent records with "bounded" and "manifest" bindings. All of the individual features of this calculus have been studied carefully in the literature, and their combination here is believed (but has not been proved!) to be sound. All of the examples in the paper have been checked mechanically by our implementation.

A.1 Syntax

The sets of kinds $\mathtt{K}$, types $\mathtt{T}$, bindings $\mathtt{B}$, terms $\mathtt{t}$, fields β, and contexts Γ are defined by the following grammer:

$\mathtt{K}$	$::=$	$*$	kind of proper types
		$\mathtt{K}\to\mathtt{K}$	operator kind
$\mathtt{T}$	$::=$	$\mathtt{X}$	type variable
		$\mathtt{Top}$	$\mathtt{Top}$ type
		$\forall[\mathtt{X}<:\mathtt{T}]\mathtt{T}$	polymorphic function type
		$\Pi[\mathtt{x}:\mathtt{T}]\mathtt{T}$	dependent function type
		$\lambda[\mathtt{X}:\mathtt{K}]\mathtt{T}$	type operator
		$\mathtt{T}\ \mathtt{T}$	type operator application
		$\{\!\vert \mathtt{l}_i \triangleright\exists \mathtt{B}_i{}^{i\in 1\cdots n} \vert\!\}$	type of records
		$\mathtt{t.l}$	record projection
$\mathtt{B}$	$::=$	$\mathtt{X}<:\mathtt{T}$	bounded type binding
		$\mathtt{X}\leftrightarrow\mathtt{T}$	manifest type binding
		$\mathtt{x}:\mathtt{T}$	term binding
$\mathtt{t}$	$::=$	$\mathtt{x}$	variable
		$\lambda[\mathtt{X}<:\mathtt{T}]\mathtt{t}$	polymorphic abstraction
		$\lambda[\mathtt{x}:\mathtt{T}]\mathtt{t}$	abstraction
		$\mathtt{t}\ \mathtt{T}$	polymorphic application
		$\mathtt{t}\ \mathtt{t}$	application
		$\{\mathtt{l}_i \triangleright\exists \beta_i{}^{i\in 1\cdots n}\}$	record intro
		$\mathtt{t.l}$	record projection
		$\mathtt{fix\ T\ t}$	fixed point operator
β	$::=$	$\mathtt{X=T}$	type field
		$\mathtt{x=t}$	term field
Γ	$::=$	$\bullet$	empty context
		$\Gamma, \mathtt{B}$	context extended by binding
		$\Gamma, \mathtt{X}:\mathtt{K}$	context extended by type binding

A.2 Abbreviations

$\mathtt{S}{\rightarrow}\mathtt{T}$	$\stackrel{\text{def}}{=}$	$\Pi\mathtt{[x:S]T}$	($\mathtt{x} \notin FV(\mathtt{T})$)
$\mathtt{Top(*)}$	$\stackrel{\text{def}}{=}$	$\mathtt{Top}$	
$\mathtt{Top(K_1{\rightarrow}K_2)}$	$\stackrel{\text{def}}{=}$	$\lambda\mathtt{[X:K_1]Top(K_2)}$	
$\forall\mathtt{[X:K]T}$	$\stackrel{\text{def}}{=}$	$\forall\mathtt{[X<:Top(K)]T}$	
$\forall\mathtt{[X]T}$	$\stackrel{\text{def}}{=}$	$\forall\mathtt{[X<:Top]T}$	
$\lambda\mathtt{[X:K]t}$	$\stackrel{\text{def}}{=}$	$\lambda\mathtt{[X<:Top(K)]t}$	
$\{\!\!\{ \ldots, \mathtt{l}\triangleright\exists\mathtt{X}, \ldots \}\!\!\}$	$\stackrel{\text{def}}{=}$	$\{\!\!\{ \ldots, \mathtt{l}\triangleright\exists\mathtt{X<:Top}, \ldots \}\!\!\}$	
$\{\!\!\{ \ldots, \mathtt{l}\triangleright\exists\mathtt{X:K}, \ldots \}\!\!\}$	$\stackrel{\text{def}}{=}$	$\{\!\!\{ \ldots, \mathtt{l}\triangleright\exists\mathtt{X<:Top(K)}, \ldots \}\!\!\}$	
$\{\!\!\{ \ldots, \mathtt{l}\triangleright\mathtt{T}, \ldots \}\!\!\}$	$\stackrel{\text{def}}{=}$	$\{\!\!\{ \ldots, \mathtt{l}\triangleright\exists\mathtt{x:T}, \ldots \}\!\!\}$	($\mathtt{x}$ not free in fields further to the right)
$\exists\triangleright\mathtt{X}$	$\stackrel{\text{def}}{=}$	$\mathtt{X}\triangleright\exists\mathtt{X}$	
$\mathtt{t::T}$	$\stackrel{\text{def}}{=}$	$(\lambda\mathtt{[x:T]x})\ \mathtt{t}$	

A.3 Reduction

$$(\lambda\mathtt{[x:T]s})\ \mathtt{t} \longrightarrow \{\mathtt{x} \mapsto \mathtt{t}\}\mathtt{s} \tag{R-Beta}$$

$$(\lambda\mathtt{[X<:S]t})\ \mathtt{T} \longrightarrow \{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{t} \tag{R-BetaT}$$

$$\frac{\beta_j = \mathtt{x=t}}{\{\mathtt{l}_i\triangleright\exists\beta_i{}^{i\in 1\cdots n}\}.\mathtt{l}_j \longrightarrow \{BV(\beta_1) \mapsto \mathtt{t.l}_1\}\cdots\{BV(\beta_{j-1}) \mapsto \mathtt{t.l}_{j-1}\}\mathtt{t}} \tag{R-Dot}$$

$$\mathtt{fix\ T\ s} \longrightarrow \mathtt{s\ (fix\ T\ s)} \tag{R-Fix}$$

(Plus congruence rules for all syntactic forms.)

A.4 Judgments

We use eight judgments of the following forms:

$\vdash \Gamma\ \mathtt{ok}$	Γ is a well-formed context
$\Gamma \vdash \mathtt{T} : \mathtt{K}$	$\mathtt{T}$ is a type constructor of kind $\mathtt{K}$
$\Gamma \vdash \mathtt{t} : \mathtt{T}$	$\mathtt{t}$ is a term of type $\mathtt{T}$
$\Gamma \vdash \beta : \mathtt{B}$	β is a record field of binding $\mathtt{B}$
$\Gamma \vdash \mathtt{S} <: \mathtt{T}$	$\mathtt{S}$ is a subtype of $\mathtt{T}$
$\Gamma \vdash \mathtt{B}_1 <: \mathtt{B}_2$	$\mathtt{B}_1$ is a subbinding of $\mathtt{B}_2$
$\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}$	$\mathtt{S}$ and $\mathtt{T}$ are equivalent types
$\Gamma \vdash \mathtt{B}_1 \leftrightarrow \mathtt{B}_2$	$\mathtt{B}_1$ and $\mathtt{B}_2$ are equivalent bindings

A.4.1 Context Well-formedness

$$\vdash \bullet\ \mathtt{ok} \tag{C-Empty}$$

$$\frac{\vdash \Gamma\ \mathtt{ok} \qquad \mathtt{X} \notin dom(\Gamma)}{\vdash \Gamma, \mathtt{X:K}\ \mathtt{ok}} \tag{C-In}$$

$$\frac{\vdash \Gamma\ \mathtt{ok} \qquad \mathtt{X} \notin dom(\Gamma) \qquad \Gamma \vdash \mathtt{T} : \mathtt{K}}{\vdash \Gamma, \mathtt{X<:T}\ \mathtt{ok}} \tag{C-Leq}$$

$$\frac{\vdash \Gamma \ \mathtt{ok} \qquad \mathtt{X} \notin dom(\Gamma) \qquad \Gamma \vdash \mathtt{T} : \mathtt{K}}{\vdash \Gamma, \mathtt{X} {\leftrightarrow} \mathtt{T} \ \mathtt{ok}} \tag{C-Eq}$$

$$\frac{\vdash \Gamma \ \mathtt{ok} \qquad \mathtt{x} \notin dom(\Gamma) \qquad \Gamma \vdash \mathtt{T} : *}{\vdash \Gamma, \mathtt{x}{:}\mathtt{T} \ \mathtt{ok}} \tag{C-Term}$$

A.4.2 Kinding Rules

$$\Gamma \vdash \mathtt{Top} : * \tag{K-Top}$$

$$\frac{\mathtt{X}{:}\mathtt{K} \in \Gamma}{\Gamma \vdash \mathtt{X} : \mathtt{K}} \tag{K-Var}$$

$$\frac{\mathtt{X} {\leftrightarrow} \mathtt{T} \in \Gamma \qquad \Gamma \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash \mathtt{X} : \mathtt{K}} \tag{K-Eq}$$

$$\frac{\mathtt{X} {<:} \mathtt{T} \in \Gamma \qquad \Gamma \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash \mathtt{X} : \mathtt{K}} \tag{K-Leq}$$

$$\frac{\vdash \Gamma, \mathtt{X}{<:}\mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{X}{<:}\mathtt{S} \vdash \mathtt{T} : *}{\Gamma \vdash \forall[\mathtt{X}{<:}\mathtt{S}]\mathtt{T} : *} \tag{K-All}$$

$$\frac{\vdash \Gamma, \mathtt{x}{:}\mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{T} : *}{\Gamma \vdash \Pi[\mathtt{x}{:}\mathtt{S}]\mathtt{T} : *} \tag{K-Pi}$$

$$\frac{\vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_n \ \mathtt{ok}}{\Gamma \vdash \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} : *} \tag{K-Rcd}$$

$$\frac{\vdash \Gamma, \mathtt{X}{:}\mathtt{K}_1 \ \mathtt{ok} \qquad \Gamma, \mathtt{X}{:}\mathtt{K}_1 \vdash \mathtt{T} : \mathtt{K}_2}{\Gamma \vdash \lambda[\mathtt{X}{:}\mathtt{K}_1]\mathtt{T} : \mathtt{K}_1 {\rightarrow} \mathtt{K}_2} \tag{K-Abs}$$

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{K}_1 {\rightarrow} \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{U} : \mathtt{K}_1}{\Gamma \vdash \mathtt{T} \ \mathtt{U} : \mathtt{K}_2} \tag{K-App}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} \qquad \mathtt{B}_j {=} \mathtt{X}{<:}\mathtt{T} \qquad \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash \mathtt{t}.\mathtt{l}_j : \mathtt{K}} \tag{K-Dot}$$

A.4.3 Typing Rules

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \tag{T-Var}$$

$$\frac{\vdash \Gamma, \mathtt{x}{:}\mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{x}{:}\mathtt{S}]\mathtt{t} : \Pi[\mathtt{x}{:}\mathtt{S}]\mathtt{T}} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Pi[\mathtt{x}{:}\mathtt{S}]\mathtt{T} \qquad \Gamma \vdash \mathtt{s} : \mathtt{S}}{\Gamma \vdash \mathtt{t} \ \mathtt{s} : \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{T}} \tag{T-App}$$

$$\frac{\vdash \Gamma, \mathtt{X}{<:}\mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{X}{<:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X}{<:}\mathtt{S}]\mathtt{t} : \forall[\mathtt{X}{<:}\mathtt{S}]\mathtt{T}} \tag{T-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall[\mathtt{X}<:\mathtt{T}]\mathtt{U} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t}\ \ \mathtt{S} : \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{U}} \tag{T-TApp}$$

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \beta_j : \mathtt{B}_j\ {}^{j\in 1\cdots n} \qquad \Gamma \vdash \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\} : *}{\Gamma \vdash \{\mathtt{l}_i {\triangleright} \exists \beta_i{}^{i\in 1\cdots n}\} : \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\}} \tag{T-Rcd}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i\in 1\cdots n} \,|\!\} \qquad \mathtt{B}_j{=}\mathtt{x}:\mathtt{T}}{\Gamma \vdash \mathtt{t.l}_j : \{BV(\mathtt{B}_1) \mapsto \mathtt{t.l}_1\} \cdots \{BV(\mathtt{B}_{j-1}) \mapsto \mathtt{t.l}_{j-1}\}\mathtt{T}} \tag{T-Dot}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}{\to}\mathtt{T}}{\Gamma \vdash \mathtt{fix}\ \ \mathtt{T}\ \ \mathtt{t} : \mathtt{T}} \tag{T-Fix}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \tag{T-Sub}$$

A.4.4 Typing Rules for Record Fields

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash (\mathtt{X}{=}\mathtt{T}) : (\mathtt{X}{\leftrightarrow}\mathtt{T})} \tag{TF-Type}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash (\mathtt{x}{=}\mathtt{t}) : (\mathtt{x}:\mathtt{T})} \tag{TF-Value}$$

A.4.5 Subtyping Rules

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \tag{S-Eqv}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash \mathtt{T} <: \mathtt{U}}{\Gamma \vdash \mathtt{S} <: \mathtt{U}} \tag{S-Trans}$$

$$\frac{\mathtt{X}<:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{X} <: \mathtt{T}} \tag{S-Leq}$$

$$\frac{\Gamma \vdash \mathtt{T} : *}{\Gamma \vdash \mathtt{T} <: \mathtt{Top}} \tag{S-Top}$$

$$\frac{\Gamma \vdash \mathtt{S}_2 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{X}<:\mathtt{S}_2 \vdash \mathtt{T}_1 <: \mathtt{T}_2 \qquad \vdash \Gamma, \mathtt{X}<:\mathtt{S}_1\ \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{X}<:\mathtt{S}_2\ \ \mathtt{ok}}{\Gamma \vdash \forall[\mathtt{X}<:\mathtt{S}_1]\mathtt{T}_1 <: \forall[\mathtt{X}<:\mathtt{S}_2]\mathtt{T}_2} \tag{S-All}$$

$$\frac{\Gamma \vdash \mathtt{S}_2 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{x}:\mathtt{S}_2 \vdash \mathtt{T}_1 <: \mathtt{T}_2 \qquad \vdash \Gamma, \mathtt{x}:\mathtt{S}_1\ \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{x}:\mathtt{S}_2\ \ \mathtt{ok}}{\Gamma \vdash \Pi[\mathtt{x}:\mathtt{S}_1]\mathtt{T}_1 <: \Pi[\mathtt{x}:\mathtt{S}_2]\mathtt{T}_2} \tag{S-Pi}$$

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{B}_j <: \mathtt{B'}_j\ {}^{j\in 1\cdots n} \qquad \vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{n+k}\ \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{B'}_1, \ldots, \mathtt{B'}_n\ \ \mathtt{ok}}{\Gamma \vdash \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B}_i{}^{i\in 1\cdots n+k} \,|\!\} <: \{\!|\, \mathtt{l}_i {\triangleright} \exists \mathtt{B'}_i{}^{i\in 1\cdots n} \,|\!\}} \tag{S-Rcd}$$

$$\frac{\vdash \Gamma, \mathtt{X}:\mathtt{K}\ \ \mathtt{ok} \qquad \Gamma, \mathtt{X}:\mathtt{K} \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X}:\mathtt{K}]\mathtt{S} <: \lambda[\mathtt{X}:\mathtt{K}]\mathtt{T}} \tag{S-Abs}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash \mathtt{S}\ \ \mathtt{U} : \mathtt{K}}{\Gamma \vdash \mathtt{S}\ \ \mathtt{U} <: \mathtt{T}\ \ \mathtt{U}} \tag{S-App}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} \qquad \mathtt{B}_j\mathtt{=X<:T}}{\Gamma \vdash \mathtt{t.l}_j \mathrel{\mathtt{<:}} \{BV(\mathtt{B}_1) \mapsto \mathtt{t.l}_1\} \cdots \{BV(\mathtt{B}_{j-1}) \mapsto \mathtt{t.l}_{j-1}\}\mathtt{T}} \quad \text{(S-Dot)}$$

A.4.6 Sub-binding Rules

$$\frac{\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}}{\Gamma \vdash (\mathtt{X<:S}) \mathrel{\mathtt{<:}} (\mathtt{X<:T})} \quad \text{(SB-Leq-Leq)}$$

$$\frac{\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}}{\Gamma \vdash (\mathtt{X}{\leftrightarrow}\mathtt{S}) \mathrel{\mathtt{<:}} (\mathtt{X<:T})} \quad \text{(SB-Eq-Leq)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{X}{\leftrightarrow}\mathtt{S}) \mathrel{\mathtt{<:}} (\mathtt{X}{\leftrightarrow}\mathtt{T})} \quad \text{(SB-Eq-Eq)}$$

$$\frac{\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}}{\Gamma \vdash (\mathtt{x:S}) \mathrel{\mathtt{<:}} (\mathtt{x:T})} \quad \text{(SB-Term-Term)}$$

A.4.7 Type Equivalence Rules

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T}} \quad \text{(TEq-Ref)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{S}} \quad \text{(TEq-Sym)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{U}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{U}} \quad \text{(TEq-Trans)}$$

$$\frac{\vdash \Gamma, \mathtt{X<:S}\ \ \mathtt{ok} \qquad \Gamma, \mathtt{X<:S} \vdash \mathtt{T} \leftrightarrow \mathtt{U}}{\Gamma \vdash \forall\mathtt{[X<:S]T} \leftrightarrow \forall\mathtt{[X<:S]U}} \quad \text{(TEq-All)}$$

$$\frac{\vdash \Gamma, \mathtt{x:S}\ \ \mathtt{ok} \qquad \Gamma, \mathtt{x:S} \vdash \mathtt{T} \leftrightarrow \mathtt{U}}{\Gamma \vdash \Pi\mathtt{[x:S]T} \leftrightarrow \Pi\mathtt{[x:S]U}} \quad \text{(TEq-Pi)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1\ \ \mathtt{T}_1 : \mathtt{K} \qquad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1\ \ \mathtt{T}_1 \leftrightarrow \mathtt{S}_2\ \ \mathtt{T}_2} \quad \text{(TEq-App)}$$

$$\frac{\Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_{j-1} \vdash \mathtt{B}_j \leftrightarrow \mathtt{B'}_j{}^{\ j \in 1 \cdots n} \qquad \vdash \Gamma, \mathtt{B}_1, \ldots, \mathtt{B}_n\ \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{B'}_1, \ldots, \mathtt{B'}_n\ \ \mathtt{ok}}{\Gamma \vdash \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} \leftrightarrow \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B'}_i{}^{i \in 1 \cdots n} \,|\!\}} \quad \text{(TEq-Rcd)}$$

$$\frac{\Gamma \vdash (\lambda\mathtt{[X:K]S})\ \ \mathtt{T} : \mathtt{K'}}{\Gamma \vdash (\lambda\mathtt{[X:K]S})\ \ \mathtt{T} \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{S}} \quad \text{(TEq-BetaT)}$$

$$\frac{\mathtt{X}{\leftrightarrow}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{X} \leftrightarrow \mathtt{T}} \quad \text{(TEq-Eq)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\!|\, \mathtt{l}_i \triangleright \exists \mathtt{B}_i{}^{i \in 1 \cdots n} \,|\!\} \qquad \mathtt{B}_j\mathtt{=X}{\leftrightarrow}\mathtt{T}}{\Gamma \vdash \mathtt{t.l}_j \leftrightarrow \{BV(\mathtt{B}_1) \mapsto \mathtt{t.l}_1\} \cdots \{BV(\mathtt{B}_{j-1}) \mapsto \mathtt{t.l}_{j-1}\}\mathtt{T}} \quad \text{(TEq-Dot)}$$

A.4.8 Binding Equivalence Rules

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{X}\mathtt{<:}\mathtt{S}) \leftrightarrow (\mathtt{X}\mathtt{<:}\mathtt{T})} \qquad \text{(BEq-Leq)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{X}{\leftrightarrow}\mathtt{S}) \leftrightarrow (\mathtt{X}{\leftrightarrow}\mathtt{T})} \qquad \text{(BEq-Eq)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash (\mathtt{x}\,\mathtt{:}\,\mathtt{S}) \leftrightarrow (\mathtt{x}\,\mathtt{:}\,\mathtt{T})} \qquad \text{(BEq-Term)}$$