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

Abstract

Virtual types have been proposed as a notation for generic programming in object-oriented languages—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 are described rather informally, usually 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, dependent functions, and dependent records with both "bounded" and "manifest" type fields (this combination of features can be viewed as a measure of the inherent complexity of virtual types). Using this model, we then discuss some of the major differences between previous proposals and show why some can be checked statically while others require run-time checks. We also investigate how the partial "duality" of virtual types and parametric classes can be understood in terms of translations between universal and (dependent) existential types.

1 Introduction

Language support for generic programming plays an important role in the development of reusable libraries. In object-oriented languages, two different approaches to genericity have been considered. The more familiar one—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* in a number of proposed extensions to Java [OW97, MBL97, AFM97, BOSW98, CS98, etc.]. An alternative approach, commonly called *virtual types* (or *virtual classes*), allows classes and objects to contain types as members, along with the usual fields and methods.[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.

Our 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 [BCP98] 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 (3) the type-theoretic intuitions behind the much-discussed "overlap" between virtual types and parametric classes in practice.

The rest of 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, for expert readers.) Section 4 develops the encoding of the Animal/Cow example in detail. Section 5 discusses the relation between virtual types and parametric classes as mechanisms for generic programming. Section 6 reviews previous work on virtual types in the light of our model. Section 7 sketches some directions for future work.

[1] Referring to this approach with the phrase "virtual types" is 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 presentation is self-contained, but somewhat technical at times. Familiarity with past work on modeling objects in typed lambda-calculi (e.g., [PT94], [HP95], [BCP98], or Chapter 18 of [AC96]) will help the reader interested in following in detail. Another useful source of background is Harper and Lillibridge [HL94, Lil97] and Leroy's [Ler94] papers on modeling module systems using dependent records with "manifest" bindings.

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 of 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 `Bag` class can be defined with a virtual type `ElementType`. Then classes `NatBag`, `StringBag`, etc. can be defined by inheriting from `Bag` and giving `ElementType` a `final` binding to `Nat` or `String`. Other examples of generic programming with virtual types can be found in [MMP89, Tho97].

3 Summary of Type System

It is well understood [PT94, BSvG95, etc.] how parametric classes—classes abstracted on type parameters—can be understood as polymorphic functions in a typed lambda-calculus. By analogy, 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 constraints on type members appearing in the interfaces `AnimalI` (`FoodType<:Food`) and `CowI` (`FoodType↔Grass`) correspond to well-known constructions in the typed lambda-calculi used by Harper and Lillibridge [HL94, Lil97] and Leroy [Ler94] to model module systems. Records with type fields constrained by <: are a generalization of *partially abstract types* [CW85]; records with type fields constrained by ↔ correspond to *translucent* or *manifest* sums.

The typed 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" [CW85] and "manifest" [HL94, Lil97, Ler94] type fields, plus dependent functions.[2] We begin by briefly reviewing the features of System $F^{\omega}_{\leq}$ (Sections 3.1 and 3.2); we then concentrate on explaining records with

[2] The system can also be described as an extension of Coquand and Huet's *Calculus of Constructions* [CH88] with subtyping and dependent records [CL96]. Experts will note that we use an unstratified presentation of dependent records; this renders the system inconsistent as a logic, but the **fix** operator does that anyway.

type fields (Section 3.3) and dependent functions (Section 3.4), which are less familiar. Appendix A gives a more formal summary of the whole system.

3.1 Functions, polymorphism, and parameterized types

The core of the system is Girard's System F^{ω} [Gir72]. This calculus can be viewed as a simple functional programming language with three distinct forms of abstraction: (1) *ordinary functions* (i.e., terms abstracted over terms); (2) *polymorphic functions* (i.e., terms abstracted over types); and (3) *parametric types* (i.e., types abstracted over types). We write all three forms with similar concrete syntax. For example,

```
plustwo = λ[x:Nat] succ(succ(x));

plustwo : Nat → Nat
```

is an ordinary function that adds two to its argument. Similarly,

```
id = λ[X:*] λ[x:X] x;

id : ∀[X] X → X
```

is the polymorphic identity function, and

```
double = λ[X:*] λ[f:X→X] λ[x:X] f(f(x));

double : ∀[X] (X→X) → X → X
```

is a polymorphic function that accepts a type `X`, a function `f` (of type `X→X`), and an argument `x` (of type `X`), and applies `f` twice to `x`. (The annotation `X:*` indicates that `X` is a type parameter.) Thus,

```
plusfour = double Nat plustwo;

plusfour : Nat → Nat
```

is a fancy way of writing the function that adds four to its (numeric) argument.

Parametric types are written in a similar style. For example,

```
Pair = λ[A:*] λ[B:*] {|fst▷A, snd▷B|};

Pair : * → * → *
```

is a convenient abbreviation for the parametric type of pairs, and

```
PairNatNat = Pair Nat Nat;

PairNatNat : *
```

is the concrete type of pairs of numbers. The usual (polymorphic) operations on pairs can be defined as follows:

```
fst = λ[A:*] λ[B:*] λ[p: Pair A B] p.fst;
snd = λ[A:*] λ[B:*] λ[p: Pair A B] p.snd;
pair = λ[A:*] λ[B:*] λ[a:A] λ[b:B]
         ({fst▷a, snd▷b} :: Pair A B);

fst : ∀[A] ∀[B] Pair A B → A
snd : ∀[A] ∀[B] Pair A B → B
pair : ∀[A] ∀[B] A → B → Pair A B
```

Note that the definition of `pair` uses an explicit coercion (`:: Pair A B`) to control how its type is printed by the typechecker. Leaving it off results in a definition with exactly the same behavior

```
pair = λ[A:*] λ[B:*]
         λ[a:A] λ[b:B] {fst▷a, snd▷b};

pair : ∀[A] ∀[B] A → B → {|fst▷A, snd▷B|}
```

(since we have defined `Pair A B` to be interchangeable with `{|fst▷A,snd▷B|}`), but less intuitive for the reader.

To ensure their well-formedness, types and type operators are assigned *kinds*, `K`, which have the form `*` or `K→K`. Type expressions of kind `*` (pronounced "type") are ordinary types; type expressions of kind `*→*` are functions from types to types; etc.

It is sometimes useful to write *higher-order* type operators—that is, type operators whose arguments are type operators. For example,

```
BothBool = λ[F:*→*→*] F Bool Bool;

BothBool : (*→*→*) → *
```

is higher-order type operator that, when applied to any operator `O`, yields `O Bool Bool`. Thus:

```
mypair = pair Bool Bool true false
           :: BothBool Pair;

mypair : BothBool Pair
```

A more natural example of higher-order type operators will be seen later in the `Object` type constructor: its argument `I` is itself an operator abstracted over the "self type" `Rep`.

For constructing objects, we shall also need a fixed-point constructor. If `t` is a function from `T` to `T`, then `fix t T` is its fixed point:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}{\to}\mathtt{T}}{\Gamma \vdash \mathtt{fix\ T\ t} : \mathtt{T}} \qquad \text{(T-FIX)}$$

For example, here is how `fix` is used to construct a factorial function:

```
fact =
  fix (Nat→Nat)
    λ[f:Nat→Nat] λ[n:Nat]
      if eq n 0
        then 1
        else times n (f (pred n));

fact : Nat → Nat
```

3.2 Subtyping

Next, we add the familiar notion of subtyping. For example, subtyping of function types is contravariant on the left and covariant on the right:

$$\frac{\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1{\to}\mathtt{S}_2 <: \mathtt{T}_1{\to}\mathtt{T}_2} \qquad \text{(S-ARROW)}$$

The subtype relation has a maximal element, called `Top`. Constraining a type variable to be a subtype of Top is actually no constraint at all, so we can recover unbounded quantification from bounded, writing λ[X<:Top]t in place of λ[X:*]t. (We will continue to write λ[X:*]t in what follows, for readability.)

Subtyping is extended *pointwise* to type operators: λ[X:K]S is a subtype of λ[X:K]T if S is a subtype of T under all legal substitutions for X.

$$\frac{\Gamma, \mathtt{X{:}K} \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \lambda\mathtt{[X{:}K]S} <: \lambda\mathtt{[X{:}K]T}} \qquad \text{(S-ABS)}$$

For example, λ[T:*] Top$\to$T is a subtype of λ[T:*] Nat$\to$T since Nat is a subtype of Top.

3.3 Records with Type Fields

To support records with type fields, a bit of 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 in dependent records.) Second, we must be able to deal with record-projection expressions like a.FoodType appearing in the types of values (e.g., a.eat). The second requirement in particular goes somewhat beyond what can be expressed using ordinary existential types, taking us into the realm of *dependent records*.

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 $\mathtt{x}_i\mathtt{=t}_i$ or a type field $\mathtt{X}_i\mathtt{=T}_i$. 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. When the internal and external names agree, we write $\exists\mathtt{x}_i\mathtt{=t}_i$ as shorthand for $\mathtt{x}_i \triangleright \exists \mathtt{x}_i\mathtt{=t}_i$, and similarly $\exists\mathtt{X}_i\mathtt{=T}_i$ for $\mathtt{X}_i \triangleright \exists \mathtt{X}_i\mathtt{=T}_i$. 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 $\mathtt{l}\triangleright\exists\mathtt{x=t}$. For example, the record value $\mathtt{r} = \{\mathtt{X}\triangleright\exists\mathtt{X=Nat}, \mathtt{l}\triangleright\exists\mathtt{x}=\lambda\mathtt{[y{:}X]y+1}\}$ is abbreviated $\{\exists\mathtt{X=Nat}, \mathtt{l}\triangleright\lambda\mathtt{[y{:}X]y+1}\}$.

(The distinction between internal and external names is discussed, for example, in [HL94]. Normally, the internal and external names of fields will agree; the possibility that they may not is mainly a technicality, which ensures that substitution and variable renaming within records behave in standard ways. Also, without the distinction, some nested records like $\{\mathtt{l}\triangleright\mathtt{X=Nat}, \mathtt{l}_1\triangleright\mathtt{x}=\{\mathtt{l}\triangleright\mathtt{Y=}\ldots, \mathtt{l}_2\triangleright\mathtt{Z=X}\}\}$ could not be expressed: the name X is needed here to refer to the field l of the outer record from the field $\mathtt{l}_2$ of the inner record—without the inner name X, this reference would be "captured" by the inner field l.)

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

Formally, the typing rule for record introduction is:

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

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 $\mathtt{x}_i\mathtt{=t}_i$ satisfy bindings of the form $\mathtt{x}_i\mathtt{:T}_i$; type fields $\mathtt{X}_i\mathtt{=T}_i$ 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 $\{\!|\, \exists\mathtt{X}\leftrightarrow\mathtt{Nat}, \mathtt{l}\triangleright\mathtt{X}\to\mathtt{X} \,|\!\}$. If we want to hide the identity of X and give r the abstract type $\{\!|\, \exists\mathtt{X{:}*}, \mathtt{l}\triangleright\mathtt{X}\to\mathtt{X} \,|\!\}$, we must 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 $\exists$x: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.[3]

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

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

The subtyping rule for record types is:

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

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 (X$\leftrightarrow$T) <: (X<:S) <: (X<:U) if T <: S <: U; the first clause ((X$\leftrightarrow$T) <: (X<:S)) allows the exact identity of a type field to be replaced with an upper bound; the second ((X<:S) <: (X<:U)), corresponding to subtyping of bounded existential types, allows us to loosen the bound of X. For example, we can derive $\{\!|\, \exists\mathtt{X}\leftrightarrow\mathtt{Nat}, \mathtt{l}\triangleright\mathtt{X}\to\mathtt{X} \,|\!\} <: \{\!|\, \exists\mathtt{X{:}*}, \mathtt{l}\triangleright\mathtt{X}\to\mathtt{X} \,|\!\}$. (As usual, this rule leads to an undecidable subtyping relation [Pie94, Lil97].)

[3] Experts will note that we give a somewhat simpler version of this rule than Harper and Lillibridge [HL94, Lil97] or Leroy's [Ler94] formulations. The reason we can do this is that we are not—at this stage—considering computational effects such as references or exceptions. If any "effectful" constructs are added to the system, our T-DOT rule need to be refined to ensure soundness. (This can be done in different ways, but the basic intuition is that a dependent projection $\mathtt{t.l}_j$ should be allowed only if the expression t is *pure*.) Similar comments apply to rule T-APP below.

3.4 Dependent Functions

For the encoding of classes, we will need to be able to give quite precise types to functions, showing the dependency of the *type* of the result on the *value* of the argument.

In outline, the intuition is this. Suppose we write a function

```
cl = λ[self:{|∃T:*, x▷T, f▷T→T|}]
       {T▷self.T, x▷self.f(self.x), f▷self.f};
```

whose argument is a record containing a type, a value (of that type), and a function (on that type), and whose result is a record with a similar shape, but where the value field is calculated by applying the argument's function field to the argument's value field. The type of this function

```
cl : Π[self: {|∃T:*,x▷T,f▷T→T|}] {|∃T↔self.T, x▷T, f▷T→T|}
```

expresses the fact that the T field of the result is identical to the T field of the argument. Next, suppose we create a record containing these three items

```
r1 = {T▷Nat, x▷3, f▷plusfour}
       :: {|∃T:*, x▷T, f▷T→T|};

r1 : {|∃T:*, x▷T, f▷T→T|}
```

and use the function cl to obtain another record of the same shape:

```
r2 = cl r1;

r2 : {|∃T↔r1.T, x▷T, f▷T→T|}
```

Notice that, because of the dependent typing of cl, the type of r2 exposes the fact that it was built from r1—in particular, that their type components are equal. Hence, it is legal to project the function field from r2 and apply it to the value field from r1:

```
i = r2.f r1.x;

i : r2.T
```

In the absence of dependent functions, the best type we could have given to cl would be:

```
cl : {|∃T:*,x▷T,f▷T→T|} → {|∃T:*, x▷T, f▷T→T|}
```

If we build r2 from r1 using this less refined type for cl,

```
r2 = cl r1;

r2 : {|∃T:*, x▷T, f▷T→T|}
```

we obtain no information about the relation between r1's T field and r2's, and the application r2.f r1.x is not allowed.

In general, a function λ[x:S]t has type Π[x:S]T, where x is allowed to appear in T. (When x does *not* appear in T, we write Π[x:S]T as S→T, recovering the usual notation for function types as a special case of dependent function types.) The rules for function abstraction and application are generalized accordingly:

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

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

4 Encoding Virtual Types

With the formalities 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 [BCP98], would yield similar results, but have not worked out the details.)

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, satiated animal rather than side-effecting the internals of the receiving animal.

In this section, we often show only an input for the typechecker and omit the output when it is trivial or obvious from explicit coercions (t::T.)

4.1 Interfaces

To get warmed up, let's begin with an example that does *not* involve virtual types: one-dimensional point objects with methods get to retrieve a current coordinate, set to move to a new coordinate, and bump to move a little from the present position.

In the simple existential encoding, the *interface* of an object is represented as a type operator of the form $\lambda[\mathtt{Rep}:*]\{|\mathtt{m}_i \triangleright \mathtt{T}_i{}^{i\in 1\cdots n}|\}$, where the bound variable Rep stands for the hidden type of the object's internal state, and where each $\mathtt{T}_i$ is type of the corresponding method $\mathtt{m}_i$. Each method takes the internal state of the object as an explicit argument and, if appropriate, returns a new internal state as its result. For example, the interface PointI of point objects is represented as

```
PointI = λ[Rep:*] {|get▷Rep→Nat,
                    set▷Rep→Nat→Rep,
                    bump▷Rep→Rep|};

PointI : * → *
```

Interfaces for objects with virtual types may include not only methods but also type fields, which declare the bounds of the virtual types. The interface AnimalI is represented as

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

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|};
```

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

4.2 Objects

Intuitively, an object with interface I comprises some hidden internal state, some methods (described by I) that can manipulate that state, and some mechanism for hiding the type of the state from outside view. In the simple existential encoding, an existential quantifier is used to achieve this hiding (it can also be done with recursive types), so the type of our point objects is:

```
Point = {|∃Rep:*,
          methods▷ {|get▷Rep→Nat,
                     set▷Rep→Nat→Rep,
                     bump▷Rep→Rep|},
          state▷ Rep|};
```

More generally, the type of objects with interface I is a record type including a representation type Rep, a method vector field containing a record of type I Rep, and a state field of type Rep. We can capture this structure uniformly by defining a (higher-order) type operator Object that takes I as a parameter:

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

Object : (*→*) → *
```

The type Point is now expressed concisely as:

```
Point = Object PointI;
```

A point object—i.e., an element of type Point—can be constructed "from scratch" as follows (we will see how to create points from classes in Section 4.3):

```
PointR = {|x▷Nat|};
point =
  {∃Rep=PointR,
   methods▷
     fix (PointI Rep)
      λ[self:PointI Rep]
        {get▷ λ[s:Rep]s.x,
         set▷ λ[s:Rep]λ[n:Nat]{x▷n},
         bump▷ λ[s:Rep] self.set s
                          (plus 1 (self.get s))},
   state▷ {x▷0}}
     :: Point;
```

PointR is the concrete representation type of the internal state. The method get just returns the x field of state, while set returns a new state with the x field set to its second argument, n. The method bump is defined in terms of the other methods get and set. In order to access other methods, the record of methods is abstracted on a parameter self of type PointI Rep; the fixed-point operator is used to "tie the knot," making self refer to the record itself.

Invocation of the get method of a Point object requires simply extracting the get field of the object's methods and applying it to the state field:

```
x = point.methods.get point.state;

x : Nat
```

More generally, we can write

```
get = λ[p:Point] p.methods.get p.state;

get : Point → Nat
```

for the function that "sends the get message" to an arbitrary point object p.

To send the set and bump messages to point objects, we need to do a little more work: the implementations of these methods return updated copies of just the internal representation, which must then be repackaged with the original methods into complete objects:

```
bump = λ[p:Point]
         {∃Rep=p.Rep,
          methods▷p.methods,
          state▷p.methods.bump p.state}
    :: Point → Point;
```

The construction of a Cow object is similar. The only significant difference is that the record of methods includes a type field FoodType, which should be given a concrete definition of food for a cow. Furthermore, methods taking arguments of FoodType can do grass-specific operation (such as enoughGrass) to the argument. Choosing the simple representation

```
CowR = {|hungry▷Bool|};
```

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

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

Like the bump method of point objects, the eatALot method of cows is defined by invoking the eat method via the self parameter.

Since we know FoodType is equal to Grass (by the definition of CowI), we can feed grass to our cow:

```
feed = λ[c:Object CowI] λ[g:Grass]
           {∃Rep=c.Rep,
            methods▷ c.methods,
            state▷ c.methods.eatALot c.state g}
        :: Object CowI → Grass → Object CowI;

satisfiedCow = feed cow grass
            :: Object CowI;
```

4.3 Classes

So far, virtual types have presented no special difficulties: the encodings of points and cows have been essentially identical. For encoding classes, however, the virtual types lead to some extra complications.

A *class* is a data structure providing implementations for a collection of methods and abstracted on a self-parameter. Concretely, a class whose instances are objects with interface I is represented as a function taking self as an argument and returning a record of methods of type I R, where R is the representation type of the state. For example, a class of point objects can be defined as follows:

```
pointClass =
  λ[self: PointI PointR]
    {get▷λ[s:PointR]s.x,
     set▷λ[s:PointR]λ[n:Nat]{x▷n},
     bump▷λ[s:PointR]self.set s
                      (plus 1 (self.get s))}
  :: PointI PointR→PointI PointR;
```

To build a point object from the point class, we choose some particular representation (some element of type PointR) and calculate its record of methods by taking the fixed point of the class:

```
point = {∃Rep=PointR,
         methods▷fix (PointI Rep) pointClass,
         state▷{x▷0}}
      :: Object PointI;
```

The fact that the methods of pointClass are abstracted on self allows us to define new subclasses of pointClass that *inherit* some of its behavior. For example, here is a class of colored point objects:

```
CPointI = λ[Rep:*] ⦃get▷Rep→Nat,
                    set▷Rep→Nat→Rep,
                    bump▷Rep→Rep,
                    color▷Rep→Color⦄;

cpointClass =
  λ[self: CPointI PointR]
    let super = pointClass self in
      {get▷super.get,
       set▷super.set,
       bump▷super.bump,
       color▷λ[s:PointR] red}
  :: CPointI PointR → CPointI PointR;

cpoint = {∃Rep=PointR,
          methods▷fix (CPointI Rep) cpointClass,
          state▷{x▷0}}
       :: Object CPointI;
```

The superclass's method suite super is obtained by application of pointClass to (cpointClass's) self. Note that, for brevity, we choose the same representation type for both pointClass and cpointClass; it is easy to generalize this so that cpointClass can add new instance variables (such as a color field), but the extra mechanism would make the examples harder to read.

When virtual types are involved, we need to be a little more precise about the typing of classes. Here, for example, is the definition of a generic animalClass. (Again, for brevity we use the same representation type (AnimalR) for both animalClass and cowClass.)

```
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⦄;
```

This definition involves a few subtle points. First, since the type FoodType and the method eat are virtual, their concrete definitions cannot be provided. Instead of concrete definitions, the corresponding fields of self are used. Second, type of animalClass is not AnimalI AnimalR→AnimalI AnimalR, but a dependent function type (a more refined subtype of AnimalI AnimalR→AnimalI AnimalR). This typing is essential when we derive cowClass from animalClass, as we will see below.

In the definition of cowClass, the FoodType and eat fields are filled with their concrete definitions and the eatALot method is inherited from animalClass. Since cowClass's self is passed to animalClass, self.eat in method eatALot refer to the eat method of cowClass (not the virtual eat method of animalClass.) Now, since FoodType is not derived from self, the type of cowClass is just a (non-dependent) function type.

```
cowClass =
  λ[self:CowI CowR]
     let super = animalClass self in
       {∃FoodType=Grass,
        eat▷λ[s:CowR]λ[f:FoodType]
            if enoughGrass f
            then {hungry▷false} else s,
        eatALot▷super.eatALot}
  :: CowI CowR → CowI CowR;
```

The dependent function type of animalClass is critical for cowClass to be well-typed: if animalClass had only type AnimalI AnimalR→AnimalI AnimalR, cowClass would be ill-typed since super.eatALot has type CowR→FoodType→CowR where FoodType <: Food, which is *not* a subtype of CowR→Grass→CowR. Thanks to the dependent function type of cowClass, super.eatALot has type CowR→self.FoodType→CowR, which is exactly equal to CowR→Grass→CowR.

Finally, a cow object can be created by instantiating cowClass in the usual way:

```
cow = {∃Rep=CowR,
       methods▷fix (CowI Rep) cowClass,
       state▷{hungry▷true}}
    :: Object CowI;
```

5 Generic Programming with Virtual Types

The "overlap" between virtual types and parametric classes as alternative mechanisms for achieving similar kinds of genericity has been remarked by several authors [BOW98, TT98, etc.]. In our encoding, this overlap can be viewed as a consequence of the inter-encodability of universal and existential polymorphism in the presence of dependent records. In this section, we first compare the two styles by means of a fully worked example, then comment on the general case.

Generic programming was one of the first applications of virtual types. The typical pattern proceeds in two steps: (1) a generic class with a virtual type is defined, with generic implementations of its operations in terms of the virtual type; (2) this class is then specialized, overriding the virtual type to some concrete instance. For example, suppose we want to program with homogeneous collections (bags) of objects of some type T. We start by building a generic Bag class with a virtual type E (which stands for type of elements) and implementations of the bag methods (put, get, etc.). Since the representation type of state of bags is parameterized by E, the interface of bags takes a type operator

Rep of kind *→*, and the type of the state is actually represented as Rep E.

```
BagI = λ[Rep:*→*] ⦃∃E:*,
                    put▷(Rep E)→E→(Rep E),
                    get▷(Rep E)→E⦄;

BagI : (*→*) → *
```

Choosing lists of elements as our internal representation,

```
BagR = λ[E:*] ⦃elts▷List(E)⦄;

BagR : * → *
```

we can define a generic bag class as follows:

```
bagClass =
    λ[self:BagI BagR]
      {∃E=self.E,
       put▷ λ[s:BagR E] λ[e:E]
              {elts▷ cons E e s.elts},
       get▷ λ[s:BagR E] car E s.elts}
    :: Π[self:BagI BagR]
         ⦃∃E↔self.E,
           put▷BagR E→E→BagR E,
           get▷BagR E→E⦄;
```

The next step is to make a subclass with a concrete definition for the element type. The class natBagClass is defined by giving the concrete value Nat to the virtual type E and by inheriting all methods from bagClass.

```
NatBagI = λ[Rep:*→*]
            ⦃∃E↔Nat,
              put▷(Rep E)→E→(Rep E),
              get▷(Rep E)→E⦄;

natBagClass =
   λ[self:NatBagI BagR]
     let super = bagClass self in
       {∃E=Nat, put▷ super.put, get▷ super.get}
   :: NatBagI BagR → NatBagI BagR;
```

The interfaces and classes here are fairly similar to the examples we saw in Section 4 (modulo the fact that the representation type here is a type operator); the construction of bag *objects*, however, requires a little explanation.

```
NatBag = ⦃∃Rep:*→*,
           ∃methods:NatBagI Rep,
           state▷Rep methods.E⦄;

natBag = {∃Rep=BagR,
          methods▷fix (NatBagI Rep) natBagClass,
          state▷ {elts▷ (nil Nat)}}
          :: NatBag;
```

The first observation is that the hidden state type is now a type operator. (Intuitively, we "see" that the representation of the object may involve the virtual type field E, but that is all we are allowed to know about the representation.) The second is that the order of the state field and the methods field is essential, since the type of the state depends both on Rep and on the E component of the methods. The code for invoking operations on bag objects is adjusted accordingly:

```
sendget = λ[b: NatBag] b.methods.get b.state
          :: NatBag → Nat;

sendput = λ[b: NatBag] λ[e:Nat]
            {∃Rep=b.Rep,
             methods▷ b.methods,
             state▷ b.methods.put b.state e}
          :: NatBag → Nat → NatBag;
```

By contrast, let's look at how bags can be modeled in terms of parametric classes. Instead of the element type being a *member* of the bag class, it will be a *parameter* to the class. Similarly, the interface BagI is parameterized by E:

```
BagI = λ[E:*] λ[Rep:*]
         ⦃put▷Rep→E→Rep, get▷Rep→E⦄;

bagClass =
    λ[E:*]
    λ[self:BagI E (BagR E)]
      {put▷ λ[s:BagR E] λ[e:E]
              {elts▷ cons E e s.elts},
       get▷ λ[s:BagR E] car E s.elts}
    :: ∀[E:*]BagI E (BagR E)→BagI E (BagR E);
```

Note that bagClass has a polymorphic function type.

The concrete instance natBagClass is now defined by *instantiating* bagClass with the type parameter Nat.

```
NatBagI = λ[Rep:*]⦃put▷Rep→Nat→Rep,
                   get▷Rep→Nat⦄;

natBagClass = bagClass Nat;
```

A bag object is defined by instantiating the class in the usual way. (Here there are no subtle dependencies between the type of the methods and state fields.)

```
NatBag = ⦃∃Rep:*,
           methods▷NatBagI Rep,
           state▷Rep⦄;

natBag = {∃Rep=BagR Nat,
          methods▷fix (NatBagI Rep) natBagClass,
          state▷ {elts▷ (nil Nat)}}
        :: NatBag;

sendget = λ[b: NatBag] b.methods.get b.state
        :: NatBag → Nat;

sendput = λ[b: NatBag] λ[e:Nat]
            {∃Rep=b.Rep,
             methods▷ b.methods,
             state▷ b.methods.put b.state e}
        :: NatBag → Nat → NatBag;
```

These examples illustrate the basic difference between virtual types and parametric classes as mechanisms for generic programming. A parametric class is instantiated by type application, taking the element type directly as an argument. With virtual types, on the other hand, type parametrization is realized by a dependent function whose argument has a type field in it. Since the get field depends on self.E, it will have type (List Nat)→Nat when the E field of the supplied self record has been set to Nat.

This correspondence can be viewed as an instance of a more general observation: polymorphic functions can be encoded in terms of dependent functions on dependent records. A polymorphic abstraction λ[X<:S]t

	$\forall$	$\exists + \Pi$
type	$\forall[X{<:}S]\ T$	$\Pi[x{:}\{\!\vert\, \exists X{<:}S \,\vert\!\}]\ (T\{X \mapsto x.X\})$
abstraction	$\lambda[X{<:}S]\ t$	$\lambda[x{:}\{\!\vert\, \exists X{<:}S \,\vert\!\}]\ (t\{X \mapsto x.X\})$
application	$t\ T$	$t\ \{\exists X{=}T\}$

Table 1: Encoding of universal types in terms of existential types

	$\exists$	$\forall$
type	$\{\!\vert\, \exists X{<:}S,\ T \,\vert\!\}$	$\forall[Y{:}*]\ (\forall[X{<:}S]T \to Y) \to Y$
packing	$\{\exists X{=}U,\ t\}$	$\lambda[Y{:}*]\ \lambda[f{:}\forall[X{<:}S]T \to Y]\ f\ U\ t$
unpacking	let $\{\exists X, y\}$ = s in (t::R)	$s\ R\ (\lambda[X{<:}S]\ \lambda[y{:}X]\ t)$

Table 2: Encoding of existential types in terms of universal types

of type $\forall[X{<:}S]T$ can be represented as the dependent function $\lambda[x{:}\{\!\vert\, \exists X{<:}S \,\vert\!\}](t\{X \mapsto x.X\})$ of type $\Pi[x{:}\{\!\vert\, \exists X{<:}S \,\vert\!\}](T\{X \mapsto x.X\})$; it takes an argument $\{\!\vert\, \exists X{\leftrightarrow}U \,\vert\!\}$ where U is some subtype of S and behaves as a term of type $T\{X \mapsto \{\!\vert\, \exists X{\leftrightarrow}U \,\vert\!\}.X\}$, which is equal to the type $T\{X \mapsto U\}$ of the corresponding polymorphic application $(\lambda[X{<:}S]t)\ U$. Table 1 summarizes this encoding. For comparison, Table 2 is the well-known encoding of (ordinary) existential types in terms of universal types.

6 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

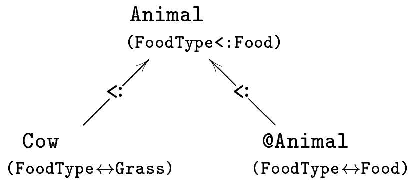
```
CowI = λ[Rep:*] ⦃∃FoodType↔Grass,
                  eat▷Rep→FoodType→Rep,
                  eatALot▷Rep→FoodType→Rep⦄;
```

as before. Finally, they want to regard cows as 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).[4]

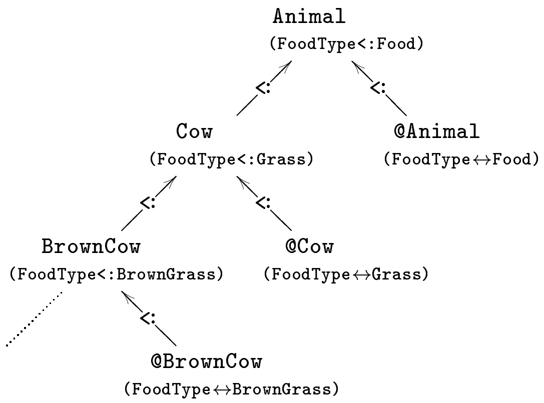
Various approaches have been suggested to remedy this unsoundness. In Beta and in Thorup's proposed Java extension, run-time checks are added to methods like eat to make sure that their arguments are actually acceptable. (In [MMMP90], it is observed that these checks can be omitted in the case where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



[4] Empirical experiments on British farms have shown that this practice leads to unsoundness.

Fortunately, the `@` variants can be derived mechanically from the other classes, as Torgersen himself pointed out in his original paper. 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 a helpful way of understanding what they did.) In their system, virtual types are 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.

Bruce, Odersky, and Wadler also pointed out that virtual types have an advantage over parametric classes in defining mutually recursive classes such as alternating lists or the Subject/Observer pattern [GHJV94]. In the Subject/Observer pattern, one object (called a *subject*) maintains a reference to a group of objects (called *observers*) so that it can notify them of "interesting" changes in its state. Using virtual types, a subject can be realized by a class with a virtual type field for the corresponding observer class, and vice versa. A generic subject class is extended to more specific subject classes—for example, window subject—by overriding the virtual observer type with the corresponding window observer class (and vice versa) and implementing specific behavior as appropriate. In [BOW98], an extension of inner classes in Java is used to define mutually recursive classes; concrete extensions (e.g., window subject/observer classes) must be defined simultaneously.

More recent work by Bruce and Vanderwaart [BV98] views virtual types as a convenient device to define mutually recursive *object types* "incrementally," by adding specifications of new methods to existing mutually recursive definitions. Since their language treats object types separately from classes, a subject class and its corresponding observer class do not have to be defined simultaneously: the virtual types in these classes will refer not to class names, but to object types.

Rémy and Vouillon [RV98] have argued that all programming idioms involving virtual types can be expressed more straightforwardly in terms of parametric classes with mutually recursive types. Since their language has not only a separate notion of object types but also type reconstruction, programmers do not even need to write object types. As we discussed in Section 5, it is not surprising that classes involving virtual types can be expressed in terms of parametric classes: the animal class, in this view, is just a parametric class taking `FoodType` as an argument. However, it is worth noting what is lost in this transformation: since `Animal` is now a parametric type, there is no longer any maximal animal type whose elements are all the animals. This means, for example, that we can build cow lists, horse lists, etc., but not animal lists.

7 Future Work

We have presented a straightforward encoding of objects with virtual types in a fairly standard (though quite powerful) type theory. We are working to extend this encoding in two main directions:

- Imperative variants of the encoding, where methods like `eat` work by side-effecting mutable instance variables.
- Recursive and mutually recursive classes involving virtual types, such as the well-known subject-observer example.

The second of these seems relatively straightforward; the first, somewhat surprisingly (and disappointingly) does not: the technicalities of the underlying type theory become astonishingly subtle when imperative features are combined with dependent types. A recent formalization of recursively defined modules by Harper, Crary, and Puri [CHP98] may be helpful in understanding and resolving these subtlties.

Another interesting question is whether the type theory in which we are working here is the simplest possible for the task. All of the features described in Section 3—in particular, both dependent records and dependent functions—are used by our encoding, but it is possible that a different encoding could get by with less.

Finally, it would be worthwhile to formalize the translation from a high-level language with virtual types into low-level structures like the ones we have explored here. The interesting point is to see how much of of the type-theoretic complexity of the target language will also show up in the typing rules of the high-level language.

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.

Discussions with Kim Bruce, Bob Harper, Didier Rémy, and Philip Wadler deepened our understanding of this material. Comments from the FOOL referees helped us improve the final presentation.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[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.

[BCP98] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*,

1998. To appear in a special issue of papers from *Theoretical Aspects of Computer Software (TACS)*, September, 1997. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, Octover 1998. ACM.

[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.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of* ECOOP '95, LNCS 952, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.

[BV98] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages, October 1998. manuscript.

[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.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[CHP98] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? Technical Report CMU–CS–98–168, Carnegie Melon University, Pittsburgh, PA, October 1998. (Also published as Fox Memorandum CMU–CS–FOX–98–03).

[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.

[CL96] Gang Chen and Giuseppe Longo. Subtyping parametric and dependent types. In Kamareddine et al., editor, *Type Theory and Term Rewriting*, September 1996. Invited lecture.

[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".

[CS98] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 33 number 10, pages 201–215, Vancouver, BC, Octover 1998. ACM.

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

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[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".

[RV98] Dedier Rémy and Jérôme Vouillon. On the (un)reality of virtual types, November 1998. manuscript.

[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: Combining the benefits of virtual types and parametrized classes, 1998. Manuscript.

A Summary of Calculus

This appendix summarizes the syntax, reduction rules, and typing rules of the type system used in this paper. 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 K, types T, bindings B, terms t, fields β, and contexts Γ are defined by the following grammar:

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

A.2 Reduction

Reduction relation $\mathtt{t} \longrightarrow \mathtt{t}'$ is the least congruence closed under the following rules.

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

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

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

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

A.3 Judgments

We use eight judgments of the following forms:

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

A.3.1 Context Well-formedness

$$\vdash \bullet\ \mathtt{ok} \qquad \text{(C-Empty)}$$

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

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

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

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

A.3.2 Kinding Rules

$$\Gamma \vdash \mathtt{Top} : * \qquad \text{(K-Top)}$$

$$\frac{\mathtt{X:K} \in \Gamma}{\Gamma \vdash \mathtt{X} : \mathtt{K}} \qquad \text{(K-Var)}$$

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

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

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

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

$$\frac{\vdash \Gamma, \mathtt{X:K}_1\ \mathtt{ok} \qquad \Gamma, \mathtt{X:K}_1 \vdash \mathtt{T} : \mathtt{K}_2}{\Gamma \vdash \lambda[\mathtt{X:K}_1]\mathtt{T} : \mathtt{K}_1 \to \mathtt{K}_2} \qquad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{K}_1 \to \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{U} : \mathtt{K}_1}{\Gamma \vdash \mathtt{T\ U} : \mathtt{K}_2} \qquad \text{(K-App)}$$

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

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

A.3.3 Typing Rules

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{\vdash \Gamma, \mathtt{X} <: \mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{X} <: \mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X} <: \mathtt{S}]\mathtt{t} : \forall[\mathtt{X} <: \mathtt{S}]\mathtt{T}} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall[\mathtt{X} <: \mathtt{T}]\mathtt{U} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t}\ \ \mathtt{S} : \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{U}} \quad \text{(T-TAPP)}$$

$$\frac{\vdash \Gamma, \mathtt{x} : \mathtt{S} \ \mathtt{ok} \qquad \Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{x} : \mathtt{S}]\mathtt{t} : \Pi[\mathtt{x} : \mathtt{S}]\mathtt{T}} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Pi[\mathtt{x} : \mathtt{S}]\mathtt{T} \qquad \Gamma \vdash \mathtt{s} : \mathtt{S}}{\Gamma \vdash \mathtt{t}\ \ \mathtt{s} : \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{T}} \quad \text{(T-APP)}$$

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

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

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \to \mathtt{T}}{\Gamma \vdash \mathtt{fix}\ \ \mathtt{T}\ \ \mathtt{t} : \mathtt{T}} \quad \text{(T-FIX)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad \text{(T-SUB)}$$

A.3.4 Typing Rules for Record Fields

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{K}}{\Gamma \vdash (\mathtt{X} = \mathtt{T}) : (\mathtt{X} \leftrightarrow \mathtt{T})} \quad \text{(TF-TYPE)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash (\mathtt{x} = \mathtt{t}) : (\mathtt{x} : \mathtt{T})} \quad \text{(TF-TERM)}$$

A.3.5 Subtyping Rules

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \quad \text{(S-EQV)}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash \mathtt{T} <: \mathtt{U}}{\Gamma \vdash \mathtt{S} <: \mathtt{U}} \quad \text{(S-TRANS)}$$

$$\frac{\mathtt{X} <: \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{X} <: \mathtt{T}} \quad \text{(S-LEQ)}$$

$$\frac{\Gamma \vdash \mathtt{T} : *}{\Gamma \vdash \mathtt{T} <: \mathtt{Top}} \quad \text{(S-TOP)}$$

$$\frac{\vdash \Gamma, \mathtt{X} <: \mathtt{S}_1 \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{X} <: \mathtt{S}_2 \ \mathtt{ok} \quad \Gamma \vdash \mathtt{S}_2 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{X} <: \mathtt{S}_2 \vdash \mathtt{T}_1 <: \mathtt{T}_2}{\Gamma \vdash \forall[\mathtt{X} <: \mathtt{S}_1]\mathtt{T}_1 <: \forall[\mathtt{X} <: \mathtt{S}_2]\mathtt{T}_2} \quad \text{(S-ALL)}$$

$$\frac{\vdash \Gamma, \mathtt{x} : \mathtt{S}_1 \ \mathtt{ok} \qquad \vdash \Gamma, \mathtt{x} : \mathtt{S}_2 \ \mathtt{ok} \quad \Gamma \vdash \mathtt{S}_2 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{x} : \mathtt{S}_2 \vdash \mathtt{T}_1 <: \mathtt{T}_2}{\Gamma \vdash \Pi[\mathtt{x} : \mathtt{S}_1]\mathtt{T}_1 <: \Pi[\mathtt{x} : \mathtt{S}_2]\mathtt{T}_2} \quad \text{(S-PI)}$$

$$\frac{\vdash \Gamma, \mathtt{X} : \mathtt{K} \ \mathtt{ok} \qquad \Gamma, \mathtt{X} : \mathtt{K} \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X} : \mathtt{K}]\mathtt{S} <: \lambda[\mathtt{X} : \mathtt{K}]\mathtt{T}} \quad \text{(S-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash \mathtt{S}\ \ \mathtt{U} : \mathtt{K}}{\Gamma \vdash \mathtt{S}\ \ \mathtt{U} <: \mathtt{T}\ \ \mathtt{U}} \quad \text{(S-APP)}$$

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

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

A.3.6 Sub-binding Rules

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

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

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

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

A.3.7 Type Equivalence Rules

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

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

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

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

$$\frac{\vdash \Gamma, \mathtt{X} <: \mathtt{S}_1 \ \mathtt{ok} \quad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma, \mathtt{X} <: \mathtt{S}_1 \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \forall[\mathtt{X} <: \mathtt{S}_1]\mathtt{T}_1 \leftrightarrow \forall[\mathtt{X} <: \mathtt{S}_2]\mathtt{T}_2} \quad \text{(TEQ-ALL)}$$

$$\frac{\vdash \Gamma, \mathtt{x} : \mathtt{S}_1 \ \mathtt{ok} \quad \Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{S}_2 \qquad \Gamma, \mathtt{x} : \mathtt{S}_1 \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_2}{\Gamma \vdash \Pi[\mathtt{x} : \mathtt{S}_1]\mathtt{T}_1 \leftrightarrow \Pi[\mathtt{x} : \mathtt{S}_2]\mathtt{T}_2} \quad \text{(TEQ-PI)}$$

$$\frac{\vdash \Gamma, \mathtt{X}{:}\mathtt{K}\ \ \mathtt{ok} \qquad \Gamma, \mathtt{X}{:}\mathtt{K} \vdash \mathtt{S} \leftrightarrow \mathtt{T}}{\Gamma \vdash \lambda[\mathtt{X}{:}\mathtt{K}]\mathtt{S} \leftrightarrow \lambda[\mathtt{X}{:}\mathtt{K}]\mathtt{T}} \qquad \text{(TEq-Abs)}$$

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

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

$$\frac{\Gamma \vdash (\lambda[\mathtt{X}{:}\mathtt{K}_1]\mathtt{S})\ \mathtt{T} : \mathtt{K}_2}{\Gamma \vdash (\lambda[\mathtt{X}{:}\mathtt{K}_1]\mathtt{S})\ \mathtt{T} \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{S}} \qquad \text{(TEq-BetaT)}$$

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

A.3.8 Binding Equivalence Rules

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

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

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