$$\frac{\Gamma \vdash t_1 : \exists X.T_{12} \qquad \Gamma, X, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } \{X,x\}=t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-UNPACK)}$$

19.2 Data Abstraction with Existentials

Abstract Data Types

For a more interesting example, here is a simple package defining an **abstract data type** of (purely functional) counters.

```
counterADT =
   {∃Counter = Nat,
    {new = 0,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};
```

```
▶ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

The concrete representation of a counter is just a number. The package provides three operations on counters: a constant `new`, a function `get` for extracting a counter's current value, and a function `inc` for creating a new counter whose stored value is one more than the given counter's. Having created the counter package, we next open it, exposing the operations as the fields of a record `counter`:

```
let {Counter,counter}=counterADT in
counter.get (counter.inc counter.new);
```

```
▶ 1 : Nat
```

If we organize our code so that the body of this `let` contains the whole remainder of the program, then this idiom

```
let {Counter,counter} = <counter package> in
<rest of program>
```

has the effect of declaring a fresh type `Counter` and a variable `counter` of type `{new:Counter,get:Counter→Nat,inc: Counter→Counter}`.

It is instructive to compare the above with a more standard abstract data type declaration, such as might be found in a program in Ada [oD80] or Clu [LAB+81]:

```
ADT counter =
  type Counter
  representation Nat
  operations
    {new = 0
         : Counter,
     get = λi:Nat. i
         : Counter→Nat,
     inc = λi:Nat. succ(i)}
         : Counter→Counter};

counter.get (counter.inc counter.new);
```

The version using existential types is somewhat harder to read, compared to the syntactically sugared second version, but otherwise the two programs are essentially identical.

Note that we can substitute an alternative implementation of the `Counter` ADT—for example, one where the internal representation is a record containing a `Nat` rather than just a single `Nat`

```
counterADT =
   {∃Counter = {x:Nat},
    {new = {x=0},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};

▶ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

in complete confidence that the whole program will remain typesafe, since we are guaranteed that the rest of the program cannot access instances of `Counter` except using `get` and `inc`. This is the essence of data abstraction by information hiding.

In the body of the program, the type name `Counter` can be used just like the base types built into the language. We can define functions that operate on counters:

```
let {Counter,counter}=counterADT
in

let addthree = λc:Counter.
                 counter.inc (counter.inc (counter.inc c))
in

counter.get (addthree counter.new);
```

```
▶ 3 : Nat
```

We can even define new abstract data types whose representation involves counters. For example, the following program defines an ADT of flip-flops, using a counter as the (not very efficient) representation type:

```
let {Counter,counter} =
     {∃Counter = Nat,
      {new = 0,
       get = λi:Nat. i,
       inc = λi:Nat. succ(i)}}
   as {∃Counter,
       {new: Counter,
        get: Counter→Nat,
        inc: Counter→Counter}}
in

let {FlipFlop,flipflop} =
     {∃FlipFlop = Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop,
        read:   FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop,
        reset:  FlipFlop→FlipFlop}}
in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
▶ false : Bool
```

19.2.1 Exercise [Recommended]: Follow the model of the above example to define an abstract data type of *stacks* of numbers, with operations `new`, `push`, `pop`, and `isempty`. Use the `List` type introduced in Exercise **??** as the underlying representation. Write a simple main program that creates a stack, pushes a couple of numbers onto it, pops off the top element, and returns it.

This exercise is best done on-line. Use the checker named "`everything`" and copy the contents of the file `test.f` from the `everything` directory (which contains definitions of the `List` constructor and associated operations) to the top of your own input file. □

Existential Objects

The sequence of "pack then open" that we saw in the last section is the hallmark of ADT-style programming using existential packages. A package defines an abstract type and its associated operations, and each package is opened immediately after it is built, binding a type variable for the abstract type and exposing the ADT's operations abstractly, with this variable in place of the concrete representation type. Existential types can also be used to model other common types of data abstraction. In this section, we show how a simple form of objects can be understood in terms of a different idiom based on existentials.

We will again use simple counters as our running example, as we did both in the previous section on existential ADTs and in our previous encounter with objects, in Chapter 14. Unlike the counters of Chapter 14, however, the counter objects in this section will be purely functional: sending the message `inc` to a counter will not change its internal state in-place, but rather will return a *fresh* counter object with incremented internal state.

A counter object, then, will comprise two basic components: a number (its internal state), and a pair of methods, `get` and `inc`, that can be used to query and update the state. We also need to ensure that the only way that the state of a counter object can be queried or updated is by using one of its two methods. This can be accomplished by wrapping the state and methods in an existential package, abstracting the type of the internal state. For example, a counter object holding the value `5` might be written

```
c = {∃X = Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
 as Counter;
```

where:

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

To use a method of a counter object, we will need to open it up and apply the appropriate element of its `methods` to its `state` field. For example, to get the current value of `c` we can write:

```
let {X,body} = c in
          body.methods.get(body.state);
▶ 5 : Nat
```

More generally, we can define a little function that "sends the `get` message" to any counter:

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
```

```
▶ sendget : Counter → Nat
```

Invoking the `inc` method of a counter object is a little more complicated. If we simply do the same as for `get`, the typechecker complains

```
 let {X,body} = c in
          body.methods.inc(body.state);
▶ Error: Scoping error!
```

because the type variable `X` appears free in the type of the body of the `let`. Indeed, what we've written doesn't make intuitive sense either, since the result of the `inc` method is a "bare" internal state, not an object. To satisfy both the typechecker and our informal understanding of what invoking `inc` should do, we must take this fresh internal state and "repackage" it as a counter object, using the same record of methods and the same internal state type as in the original object:

```
c1 = let {X,body} = c in
       {∃X = X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
 as Counter;
```

More generally, to "send the `inc` message" to an arbitrary counter object, we can write:

```
sendinc = λc:Counter.
            let {X,body} = c in
              {∃X = X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter
```

More complex operations on counters can be implemented in terms of these two basic operations:

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▶ addthree : Counter → Counter
```

19.2.2 Exercise: Implement `FlipFlop` objects with `Counter` objects as their internal representation type, following the model of the `FlipFlop` ADT in Section 19.2. □

Objects vs. ADTs

What we have seen in Section 19.2 falls significantly short of a full-blown model of object-oriented programming. Many of the features that we saw in Chapter 14, including subtyping, classes, inheritance, and recursion through `self` and `super`, are missing here. We will come back to modeling these features in later chapters, when we have added a few necessary refinements to our modeling language. But even for the simple objects we have developed so far, there are several interesting comparisons to be made with ADTs.

At the coarsest level, the two programming idioms fall at opposite ends of a spectrum: when programming with ADTs, packages are opened immediately after they are built; on the other hand, when packages are used to model objects they are kept closed as long as possible—until the moment when they *must* be opened so that one of the methods can be applied to the internal state.

A consequence of this difference is that the "abstract type" of counters refers to different things in the two styles. In an ADT-style program, the counter values manipulated by client code such as `addthree` are elements of the underlying representation type (e.g., simple numbers). In an object-style program, a counter value is a whole package—not only a number, but also the implementations of the `get` and `inc` methods. This stylistic difference is reflected in the fact that, in the ADT style, the type `Counter` is a bound type variable introduced by the `let` construct, while in the object style `Counter` abbreviates the whole existential type $\{\exists X, \{state:X, methods: \{get:X\to Nat, inc:X\to X\}\}\}$. Thus:

- All the counter values generated from the counter ADT are elements of the same internal representation type; there is a single implementation of the counter operations that works on this internal representation.
- Each counter object, on the other hand, carries its own representation type and its own set of methods that work for this representation type.

19.2.3 Exercise: In what ways do the *classes* found in mainstream object-oriented languages like C++ and Java resemble the simple object types discussed here? In what ways do they resemble ADTs? □

19.3 Encoding Existentials

The encoding of pairs as a polymorphic type in Exercise 18.4.5 suggests a similar encoding for existential types in terms of universal types, using the intuition that an element of an existential type is a pair of a type and a value:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X.\ T\to Y) \to Y.$$

That is, an existential package is thought of as a data value that, given a result type and a "continuation," calls the continuation to yield a final result. The continuation

takes two arguments—a type X and a value of type T—and uses them in computing the final result.

Given this encoding of existential types, the encoding of the packaging and unpackaging constructs is essentially forced. To encode a package

```
{∃X=S,t} as {∃X,T}
```

we must exhibit a value of type ∀Y. (∀X. T→Y) → Y. This type begins with a universal quantifier, the body of which is an arrow. An element of this type should therefore begin with two abstractions:

```
{∃X=S,lt} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). ...
```

To complete the job, we need to return a result of type Y; clearly, the only way to do this is to apply f to some appropriate arguments. First, we supply the type S (this is a natural choice, being the only type we have lying around at the moment):

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] ...
```

Now, the type application f [S] has type {X ↦ S}(T→Y), i.e., ({X ↦ S}T)→Y. We can thus supply t (which, by rule T-PACK, has type {X ↦ S}T) as the next argument:

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] t
```

The type of the whole application f [S] t is now Y, as required.

To encode the unpacking construct

```
let {X,x}=t1 in t2  ≝  ...
```

we proceed as follows. First, the typing rule T-UNPACK tells us that t_1 should have some type {∃X,T_{11}}, that t_2 should have type T_2 (under an extended context binding X and x:T_{11}), and that T_2 is the type we expect for the whole let...in... expression.[2] As in the Church encodings in Section 18.4, the intuition here is that the introduction form ({∃X=S,t}) is encoded as an active value that "performs its own elimination." So the encoding of the elimination form here should simply take the existential package t_1 and apply it to enough arguments to yield a result of the desired type T_2:

```
let {X,x}=t1 in t2  ≝  t1 ...
```

The first argument to t_1 should be the desired result of the whole expression, i.e., T_2:

```
let {X,x}=t1 in t2  ≝  t1 [T2] ...
```

[2] Strictly speaking, the fact that the translation requires these extra bits of type information not present in the syntax of terms means that what we are translating is actually *typing derivations*, not terms.

Now, the application $t_1\ [T_2]$ has type $(\forall X.\ T{\to}T_2) \to T_2$. That is, if we can now supply another argument of type $(\forall X.T{\to}T_2)$, we will be finished. Such an argument can be obtained by *abstracting* the body t_2 on the variables `X` and `x`:

$$\texttt{let } \{X,x\}{=}t_1 \texttt{ in } t_2 \stackrel{\text{def}}{=} t_1\ [T_2]\ (\lambda X.\ \lambda x{:}T_{11}.\ t_2).$$

This finishes the encoding.

19.3.1 Exercise: What must we prove to show that our encoding of existentials is correct? □

19.3.2 Exercise [Recommended]: Take a blank piece of paper and, without looking at the above encoding, regenerate it from scratch. □

19.3.3 Exercise: Can universal types be encoded in terms of existential types? □

19.4 Implementation

```
type ty =
    ...
  | TySome of string * ty

type term =
    ...
  | TmPack of info * string * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

19.5 Historical Notes

The correspondence between ADTs and existential types was first developed by Mitchell and Plotkin [MP88]. (They also noticed the correspondance with objects.)

Chapter 20

Bounded Quantification

Many of the interesting problems in type systems arise from the *combination* of features that, in themselves, may be quite simple. In this chapter, we encounter our first substantial example: a system that mixes subtyping with polymorphism.

The most basic combination of these features is actually quite straightforward. We simply add to the subtyping relation a rule for comparing quantified types:

$$\frac{S <: T}{\forall X.S <: \forall X.T}$$

We consider here a more interesting combination, in which the syntax, typing, and subtyping rules for universal quantifiers are actually refined to take subtyping into account. The resulting notion of **bounded quantification** substantially increases both the expressive power of the system and its metatheoretic complexity.

20.1 Motivation

To see why we might want to combine subtyping and polymorphism in this more intimate manner, consider the identity function on records with a numeric field `a`:

```
f = λx:{a:Nat}. x;
▶ f : {a:Nat} → {a:Nat}
```

If we define a record of this form

```
ra = {a=0};
```

then we can apply `f` to `ra` (in any of the systems that we have seen), yielding a record of the same form.

```
(f ra);
▶ {a=0} : {a:Nat}
```

If we define a larger record `rab` with two fields, `a` and `b`,

```
rab = {a=0, b=true};
```

we can also apply `f` to `rab`, using the rule of subsumption introduced in Chapter 13.

```
(f rab);
▶ {a=0, b=true} : {a:Nat}
```

However, the type of the result has only the field `a`, which means that a term like `(f rab).b` will be judged ill typed. In other words, by passing `rab` through the identity function, we have lost the ability to access its `b` field!

Using the polymorphism of System F, we can write `f` in a different way:

```
fpoly = λX. λx:X. x;
▶ fpoly : ∀X. X → X
```

The application of `fpoly` to `rab` (and an appropriate type argument) yields the desired result:

```
(fpoly [{a:Nat, b:Bool}] rab);
▶ {a=0, b=true} : {a:Nat, b:Bool}
```

But in making the type of `x` into a variable, we have given up some information that `f` might have wanted to use. For example, suppose we intend that `f` return a pair of its original argument and the numeric successor of its `a` field.

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▶ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

Again, we can apply `f2` to both `ra` and `rab`, losing the `b` field in the second case.

```
(f2 ra);
▶ {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}

(f2 rab);
▶ {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

But this time polymorphism offers us no solution. If we replace the type of `x` by a variable `X` as before, we lose the constraint that `x` must have an `a` field, which is required to compute the `asucc` field of the result.

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▶ Error: Expected record type
```

The fact about the operational behavior of `f2` that we want to express in its type is:

> `f2` takes an argument of any record type `R` that includes a numeric `a` field and returns as its result a record containing a field of type `R` and a field of type `Nat`.

We can use the subtype relation to express this concisely as follows:

> `f2` takes an argument of any subtype `R` of the type `{a:Nat}` and returns a record containing a field of type `R` and a field of type `Nat`.

This intuition can be formalized by introducing a **subtyping constraint** on the bound variable `X` of `f2poly`.

```
f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▶ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

This interaction of subtyping and polymorphism, called **bounded quantification**, leads us to a type system commonly called System $F_{<:}$ ("F sub"), which is the topic of this chapter.

20.2 Definitions

We form System $F_{<:}$ by combining the types and terms of System F with the subtype relation from Chapter 13 and refining universal quantifiers with subtyping constraints on their bound variables. When we define the subtyping rule for these bounded quantifiers, there will actually be two choices: a more tractable but less flexible rule called the **kernel** rule and a more expressive **full** subtyping rule, which will turn out to raise some unexpected difficulties when we come to designing typechecking algorithms.

Kernel $F_{<:}$

Since type variables now have associated bounds (just as ordinary variables have associated types), we must keep track of them during both subtyping and typechecking. We change the type bindings in contexts to include an upper bound for each type variable, and add contexts to all the rules in the subtype relation. These bounds will be used during subtyping to justify steps of the form "the type variable `X` is a subtype of the type `T` because we assumed it was."

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVAR)}$$

20.2.1 Exercise [Quick check]: Exhibit a subtyping derivation showing that

$$B<:\mathtt{Top},\ X<:B,\ Y<:X \vdash B\to Y <:\ X\to B.$$

□

Next, we introduce bounded universal types, extending the syntax and typing rules for ordinary universal types in the obvious way. The only rule where the extension is not completely obvious is the subtyping rule for quantified types, S-ALL. We give here the simpler variant, called the **kernel** subtyping rule for universal quantifiers, in which the bounds of the two quantifiers being compared must be identical. (The term "kernel" comes from Cardelli and Wegner's original paper [CW85], where this variant of $F_{<:}$ was called **Kernel Fun**.)

$$\frac{\texttt{X<:T} \in \Gamma}{\Gamma \vdash \texttt{X} <: \texttt{T}} \qquad \text{(S-TVAR)}$$

For easy reference, here is the complete definition of kernel $F_{<:}$, with differences from previous systems highlighted:

$F^k_{<:}$: Bounded quantification $\rightarrow \forall <: bq$

Syntax

t ::=		(*terms...*)
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX<:T.t	*type abstraction*
	t [T]	*type application*
v ::=		(*values...*)
	λx:T.t	*abstraction value*
	λX<:T.t	*type abstraction value*
T ::=		(*types...*)
	X	*type variable*
	Top	*maximum type*
	T→T	*type of functions*
	∀X<:T.T	*universal type*
Γ ::=		(*contexts...*)
	∅	*empty context*
	Γ, x:T	*term variable binding*
	Γ, X<:T	*type variable binding*

Evaluation $(t \longrightarrow t')$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda X <: T_{11}.t_{12})\ [T_2] \longrightarrow \{X \mapsto T_2\}t_{12} \quad \text{(E-Beta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

Subtyping ($\Gamma \vdash S <: T$)

$$\Gamma \vdash S <: S \quad \text{(S-Refl)}$$

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

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

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

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1.S_2 <: \forall X <: U_1.T_2} \quad \text{(S-All)}$$

Typing ($\Gamma \vdash t : T$)

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

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1.t_2 : \forall X <: T_1.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}T_{12}} \quad \text{(T-TApp)}$$

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

Full $F_{<:}$

In kernel $F_{<:}$, two quantified types can only be compared if their upper bounds are identical. If we think of quantifiers as a kind of arrow types (since they classify functions from types to terms), then the kernel rule corresponds to a "covariant" version of the subtyping rule for arrows, in which the domain of an arrow type is not allowed to vary in subtypes:

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

This restriction feels rather unnatural, both for arrows and for quantifiers. Carrying the analogy a little further, we can allow the "left-hand side" of bounded quantifiers to vary (contravariantly) during subtyping:

$F^f_{<:}$: "Full" bounded quantification $\to \forall <: bq\ full$

New subtyping rules ($\Gamma \vdash S <: T$)

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1.S_2 <: \forall X<:T_1.T_2} \quad \text{(S-ALL)}$$

Intuitively, the "full $F_{<:}$" quantifier subtyping rule can be understood as follows. A type $T = \forall X<:T_1.T_2$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of T_1 to instances of T_2. If T_1 is a subtype of S_1, then the domain of T is smaller than that of $S = \forall X<:S_1.S_2$, so S is a stronger constraint and describes a smaller collection of polymorphic values. Moreover, if, for each type U that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $U <: T_1$), the U-instance of S_2 is a subtype of the U-instance of T_2, then S is a "pointwise stronger" constraint and again describes a smaller collection of polymorphic values.

The system with just the kernel subtyping rule for quantified types is called Kernel $F_{<:}$ (or $F^k_{<:}$). The same system with the full quantifier subtyping rule is called Full $F_{<:}$ (or $F^f_{<:}$). The bare name $F_{<:}$ refers ambiguously to both systems.

20.2.2 Exercise [Quick check]: Give a couple of examples of pairs of types that are related by the subtype relation of full $F_{<:}$ but are not subtypes in kernel $F_{<:}$. □

20.2.3 Exercise [Challenging]: Can you find any *useful* examples with this property? □

20.3 Examples

We now present some simple examples of programming in $F_{<:}$. More sophisticated uses of bounded quantification will appear in later chapters.

Encoding Products

In Exercise **??**, we gave the following encoding of pairs in System F. The elements of the type

```
Pair T1 T2 = ∀X. (T1→T2→X) → X;
```

correspond to pairs of T_1 and T_2. The constructor `pair` and the destructors `fst` and `snd` were defined as follows:

```
pair = λX. λY. λx:X. λy:Y.
         (   λR. λp:X→Y→R. p x y
          as Pair X Y);

fst = λX. λY. λp: Pair X Y.
        p [X] (λx:X. λy:Y. x);

snd = λX. λY. λp: Pair X Y.
        p [Y] (λx:X. λy:Y. y);
```

Of course, the same encoding can be used in $F_{<:}$, since $F_{<:}$ contains all the features of System F. What is interesting, though, is that this encoding also has some natural subtyping properties. In fact, the expected subtyping rule for pairs

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \texttt{Pair}\ S_1\ S_2 <: \texttt{Pair}\ T_1\ T_2}$$

follows directly from the encoding.

20.3.1 Exercise [Quick check]: Show this. □

Encoding Records

It is interesting to notice that records and record types—including subtyping—can actually be encoded in the pure calculus. The encoding presented here was discovered by Cardelli [Car92]. We begin by defining **flexible tuples** as follows:

20.3.2 Definition: For each $n \geq 0$ and types T_1 through T_n, let

$$\{T_{i\ i\in 1..n}\} \stackrel{\text{def}}{=} \texttt{Pair}\ T_1\ (\texttt{Pair}\ T_2\ \ldots\ (\texttt{Pair}\ T_n\ \texttt{Top})\ldots).$$

In particular, $\{\} = \texttt{Top}$. Similarly, for terms t_1 through t_n, let

$$\{t_{i\ i\in 1..n}\} \stackrel{\text{def}}{=} \texttt{pair}\ t_1\ (\texttt{pair}\ t_2\ \ldots\ (\texttt{pair}\ t_n\ \texttt{top})\ldots),$$

where we elide the type arguments to `pair`, for the sake of brevity. (Recall that `top` is just some element of `Top`.) The projection `t.n` (again eliding type arguments) is:

$$\texttt{fst}(\underbrace{\texttt{snd}(\texttt{snd}\ldots(\texttt{snd}}_{n-1 \text{ times}}\texttt{t})\ldots) \qquad \square$$

From this abbreviation, we immediately obtain the following rules for subtyping and typing.

$$\frac{\Gamma \vdash^{i\in 1..n} S_i <: T_i}{\Gamma \vdash \{S_i{}^{i\in 1..n+k}\} <: \{T_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash^{i\in 1..n} t_i : T_i}{\Gamma \vdash \{t_i{}^{i\in 1..n}\} : \{T_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{T_i{}^{i\in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

Now, let $\mathcal{L}$ be a countable set of labels, with a fixed total ordering given by the bijective function *label-with-index* : $\mathbb{N} \to \mathcal{L}$. We define records as follows:

20.3.3 Definition: Let L be a finite subset of $\mathcal{L}$ and let S_l be a type for each $l \in L$. Let m be the maximal index of any element of L, and

$$\hat{S}_i = \begin{cases} S_l & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{Top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record type $\{l{:}S_l{}^{l\in L}\}$ is defined as the flexible tuple $\{\hat{S}_i{}^{i\in 1..m}\}$. Similarly, if t_l is a term for each $l : L$, then

$$\hat{t}_i = \begin{cases} t_i & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record value $\{l{=}t_l{}^{l\in L}\}$ is $\{\hat{t}_i{}^{i\in 1..m}\}$. The projection $t.l_i$ is just the tuple projection `t.i`. $\square$

This encoding validates the expected rules for typing and subtyping:

$$\Gamma \vdash \{l_i{:}T_i{}^{i\in 1..n+k}\} <: \{l_i{:}T_i{}^{i\in 1..n}\} \qquad \text{(S-RCD-WIDTH)}$$

$$\frac{\text{for each } i \quad \Gamma \vdash S_i <: T_i}{\Gamma \vdash \{l_i{:}S_i{}^{i\in 1..n}\} <: \{l_i{:}T_i{}^{i\in 1..n}\}} \qquad \text{(S-RCD-DEPTH)}$$

Church Encodings with Subtyping

As a last simple illustration of the expressiveness of $F_{<:}$, let's take a look at what happens when we add bounded quantification to the encoding of Church Numerals in System F that we saw in Section 18.4. The original polymorphic type of church numerals was:

```
CNat = ∀X. (X→X) → X → X;
```

The intuitive reading of this type was: "Tell me a result type `T` and give me a function on `T` and an element of `T`, and I'll give you back another element of `T` formed by iterating the function you gave me `n` times over the base value you gave."

We can generalize this by adding two bounded quantifiers and refining the types of the parameters `s` and `z`.

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

Intuitively, this type can be read as follows: "Give me a generic result type `T` and two subtypes `S` and `Z`. Then give me a function that maps from the whole set `T` into the subset `S` and an element of the special set `Z`, and I'll return you an element of `T` formed in the same way as before."

To see why this is an interesting generalization, consider this slightly different type:

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
```

Although `SZero` has almost the same form as `SNat`, it says something much stronger about the behavior of its elements, since it promises that its result will be an element of `Z`, not just of `T`. In fact, there is just one way that an element of `Z` could be returned—namely by yielding just `z` itself. In other words, the value

```
szero = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z) as SZero;
▶ szero : SZero
```

is the *only* inhabitant of the type `SZero`. On the other hand, the similar type

```
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

has more inhabitants; for example,

```
sone   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z) as SPos;
stwo   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z)) as SPos;
sthree = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z))) as SPos;
```

and so on.

Moreover, notice that `SZero` and `SPos` are both subtypes of `SNat` (Exercise: check this), so we also have `szero` ∶ `SNat`, `sone` ∶ `SNat`, `stwo` ∶ `SNat`, etc.

Finally, we can similarly refine the typings of operations defined on church numerals. For example, the type system is capable of detecting that the successor function always returns a positive number:

```
ssucc = λn:SNat.
          (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
            s (n [X] [S] [Z] s z))
          as SPos;
```

```
▶ ssucc : SNat → SPos
```

Similarly, by refining the types of its parameters, we can write the function `plus` in such a way that the typechecker gives it the refined type SPos→SZero→SPos.

```
spluspz = λn:SPos. λm:SZero.
             (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
               n [X] [S] [Z] s (m [X] [S] [Z] s z))
             as SPos;
```

```
▶ spluspz : SPos → SZero → SPos
```

20.3.4 Exercise: Write a similar version of `plus` that has type SPos→SPos→SPos. Write one that has type SNat→SNat→SNat. □

The previous example and exercise raise an interesting point: obviously, we don't want to have several different versions of `plus` lying around and have to decide which to apply based on the expected types of its arguments: we want to have a *single* version of `plus` whose type contains all these possibilities—something like

```
plus :   SZero→SZero→SZero
       ∧ SNat→SPos→SPos
       ∧ SPos→SNat→SPos
       ∧ SNat→SNat→SNat
```

The desire to support this kind of overloading has led to the study of systems with **intersection types**.

20.3.5 Exercise [Recommended]: Generalize the type `CBool` of Church Booleans from Section 18.4 in a similar way by defining a type `SBool` and two subtypes `STrue` and `SFalse`. Write a function `notft` with type SFalse→STrue and a similar one `nottf` with type STrue→SFalse. □

The examples that we have seen in this section are amusing to play with, but they might not convince you that $F_{<:}$ is a system of tremendous practical importance! We will come to some more interesting uses of bounded quantification in Chapter 28, but these will require just a little more machinery, which we will develop in the intervening chapters.

20.4 Safety

We now consider the metatheory of both kernel and full systems of bounded quantification ($F^k_{<:}$ and $F^f_{<:}$). Much of the development is the same for both systems: we carry it out first for the simpler case of $F^k_{<:}$ and then consider $F^f_{<:}$.

The type preservation property can actually be proved quite directly for both systems, with minimal technical preliminaries. This is good, since the soundness of the type system is a critical property, while other properties such as decidability may be less important in some contexts. (The soundness theorem belongs in the language definition, while decision procedures are buried in the compiler.) We develop the proof in detail for $F^k_{<:}$. The argument for $F^f_{<:}$ is very similar.

We begin with a couple of technical facts about the typing and subtyping relations. The proofs go by straightforward induction on derivations.

20.4.1 Lemma [Permutation]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{S} <: \mathtt{T}$. □

20.4.2 Lemma [Weakening]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{X} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{X}<:\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
3. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$.
4. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{X} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{X}<:\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$. □

As usual, the proof of type preservation relies on several lemmas relating substitution with the typing and subtyping relations.

20.4.3 Definition: We write $\{\mathtt{X} \mapsto \mathtt{S}\}\Gamma$ for the context obtained by substituting $\mathtt{S}$ for $\mathtt{X}$ in the right-hand sides of all of the bindings in Γ. □

20.4.4 Exercise [Quick check]: Show the following properties of subtyping and typing derivations:

1. if $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \mathtt{X}<:\mathtt{P}, \Delta \vdash \mathtt{S} <: \mathtt{T}$;
2. if $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \mathtt{X}<:\mathtt{P}, \Delta \vdash \mathtt{t} : \mathtt{T}$;

These properties are often called **narrowing** because they involve restricting the range of the variable $\mathtt{X}$. □

Next, we have the usual lemma relating substitution and the typing relation.

20.4.5 Lemma [Substitution preserves typing]: If $\Gamma, \mathtt{x}:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{q} : \mathtt{Q}$, then $\Gamma, \Delta \vdash \{\mathtt{x} \mapsto \mathtt{q}\}\mathtt{t} : \mathtt{T}$. □

Proof: Straightforward induction on a derivation of $\Gamma, \mathtt{x}:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$, using the properties proved above. □

Since we may substitute types for type variables during reduction, we also need a lemma relating type substitution and typing, as we did in System F. Here, though, we must deal with one new twist: the proof of this lemma (specifically, the T-SUB case) depends on a new lemma relating substitution and subtyping:

20.4.6 Lemma [Type substitution preserves subtyping]: If $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$. The only interesting cases are the last two:

Case S-TVAR: $\mathtt{S} = \mathtt{Y}$ $\mathtt{Y}<:\mathtt{T} \in (\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta)(\mathtt{Y})$

There are two subcases to consider. If $\mathtt{Y} \neq \mathtt{X}$, then the result follows immediately from S-TVAR. On the other hand, if $\mathtt{Y} = \mathtt{X}$, then we have $\mathtt{T} = \mathtt{Q}$ and $\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} = \mathtt{Q}$, and the result follows by S-REFL.

Case S-ALL: $\mathtt{S} = \forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{S}_2$ $\mathtt{T} = \forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{T}_2$
$\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta, \mathtt{Z}<:\mathtt{U}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta, \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1 \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$. By S-ALL, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$, that is, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{S}_2) <: \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{T}_2)$, as required. □

20.4.7 Lemma [Type substitution preserves typing]: If $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$. We give just the interesting cases.

Case T-TAPP: $\mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2]$ $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t}_1 : \forall \mathtt{Z}<:\mathtt{T}_{11}.\mathtt{T}_{12}$
$\mathtt{T} = \{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{T}_{11}.\mathtt{T}_{12})$, i.e, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \forall \mathtt{Z}<:\mathtt{T}_{11}.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12}$. By T-TAPP, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1\ [(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2]) : \{\mathtt{Z} \mapsto \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2\}(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12})$, i.e., $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\mathtt{t}_1\ [\mathtt{T}_2]) : \{\mathtt{X} \mapsto \mathtt{P}\}(\{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12})$.

Case T-SUB: $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{S}$ $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. By the preservation of subtyping under substitution (20.4.6), $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$, and the result follows by T-SUB. □

Next, we establish some simple structural facts about the subtype relation.

20.4.8 Lemma [Inversion of the subtyping relation, from right to left]:

1. If $\Gamma \vdash S <: X$, then S is a type variable.
2. If $\Gamma \vdash S <: T_1 \to T_2$, then either S is a type variable or else $S = S_1 \to S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$.
3. If $\Gamma \vdash S <: \forall X<:U_1.T_2$, then either S is a type variable or else $S = \forall X<:U_1.S_2$ with $\Gamma, X<:U_1 \vdash S_2 <: T_2$. □

Proof: Part (1) follows by an easy induction on subtyping derivations. The only interesting case is the rule S-TRANS, which proceeds by two uses of the induction hypothesis, first on the right premise and then on the left. The arguments for the other parts are similar (part (1) is used in the transitivity cases). □

20.4.9 Exercise: Show the following "left to right inversion" properties:

1. If $\Gamma \vdash S_1 \to S_2 <: T$, then either $T = \mathrm{Top}$ or else $T = T_1 \to T_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$.
2. If $\Gamma \vdash \forall X<:U.S_2 <: T$, then either $T = \mathrm{Top}$ or else $T = \forall X<:U.T_2$ with $\Gamma, X<:U \vdash S_2 <: T_2$.
3. If $\Gamma \vdash X <: T$, then either $T = \mathrm{Top}$ or $T = X$ or $\Gamma \vdash S <: T$, where $X<:S \in \Gamma$.
4. If $\Gamma \vdash \mathrm{Top} <: T$, then $T = \mathrm{Top}$. □

We use Lemma 20.4.8 for one straightforward structural property of the typing relation that will be needed in the critical cases of the type preservation proof.

20.4.10 Lemma:

1. If $\Gamma \vdash \lambda x:S_1.s_2 : T$ and $\Gamma \vdash T <: U_1 \to U_2$, then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x:S_1 \vdash s_2 : S_2$ and $\Gamma \vdash S_2 <: U_2$.
2. If $\Gamma \vdash \lambda X<:S_1.s_2 : T$ and $\Gamma \vdash T <: \forall X<:U_1.U_2$, then $U_1 = S_1$ and there is some S_2 such that $\Gamma, X<:S_1 \vdash s_2 : S_2$ and $\Gamma, X<:S_1 \vdash S_2 <: U_2$. □

Proof: Straightforward induction on typing derivations, using Lemma 20.4.8 for the induction case (rule T-SUB). □

With all these facts in-hand, the actual proof of type preservation is straightforward.

20.4.11 Theorem [Preservation]: If $\Gamma \vdash t : T$ and $\Gamma \vdash t \longrightarrow t'$, then $\Gamma \vdash t' : T$. □

Proof: By induction on a derivation of $\Gamma \vdash t : T$. All of the cases are straightforward, using the facts established in the above lemmas.

Case T-VAR: $t = x$

This case cannot actually arise, since we assumed $\Gamma \vdash t \longrightarrow t'$ and there are no evaluation rules for variables.

Case T-ABS: $t = \lambda x{:}T_1.t_2$

Ditto.

Case T-APP: $t = t_1\ t_2 \qquad \Gamma \vdash t_1 : T_{11}{\to}T_{12}$
$T = T_{12} \qquad \Gamma \vdash t_2 : T_{11}$

By the definition of the evaluation relation, there are three subcases to consider:

Subcase: $\Gamma \vdash t_1 \longrightarrow t_1' \qquad t' = t_1'\ t_2$

Then the result follows from the induction hypothesis and T-APP.

Subcase: t_1 is a value $\qquad \Gamma \vdash t_2 \longrightarrow t_2' \qquad t' = t_1\ t_2'$

Similar.

Subcase: $t_1 = \lambda x{:}U_{11}.u_{12} \qquad t' = \{x \mapsto t_2\}u_{12}$

By Lemma 20.4.10, $\Gamma, x{:}U_{11} \vdash u_{12} : U_{12}$ with $\Gamma \vdash T_{11} <: U_{11}$ and $\Gamma \vdash U_{12} <: T_{12}$. By the preservation of typing under substitution (Lemma 20.4.5), $\Gamma \vdash \{x \mapsto t_2\}u_{12} : U_{12}$, from which $\Gamma \vdash \{x \mapsto t_2\}u_{12} : T_{12}$ follows by T-SUB.

Case T-TABS: $t = \lambda X{<:}U.t$

Can't happen.

Case T-TAPP: $t = t_1\ [T_2] \qquad \Gamma \vdash t : \forall X{<:}T_{11}.T_{12}$
$T = \{X \mapsto T_2\}T_{12} \qquad \Gamma \vdash T_2 <: T_{11}$

By the definition of the evaluation relation, there are two subcases to consider:

Subcase: $t_1 \longrightarrow t_1' \qquad t' = t_1'\ [T_2]$

The result follows from the induction hypothesis and T-TAPP.

Subcase: $t_1 = \lambda X{<:}U_{11}.u_{12} \qquad t' = \{X \mapsto T_2\}u_{12}$

By Lemma 20.4.10, $U_{11} = T_{11}$ and $\Gamma, X{<:}U_{11} \vdash u_{12} : U_{12}$ with $\Gamma, X{<:}U_{11} \vdash U_{12} <: T_{12}$. By the preservation of typing under substitution (20.4.5), $\Gamma \vdash \{X \mapsto T_2\}u_{12} : \{X \mapsto T_2\}U_{12}$, from which $\Gamma \vdash \{X \mapsto T_2\}u_{12} : \{X \mapsto T_2\}T_{12}$ follows by Lemma 20.4.6 and T-SUB.

Case T-SUB: $\Gamma \vdash t : S \qquad \Gamma \vdash S <: T$

By the induction hypothesis, $\Gamma \vdash t' : S$; the result follows by T-SUB. □

20.4.12 Exercise: Show how to extend the argument in this section to $F^f_{<:}$. □

20.5 Bounded Existential Types

This final section remains to be written.

Bounded existential quantification $F^k_{<:} + \exists$

New syntactic forms

T ::=	...	(*types...*)
	$\{\exists X <: T, T\}$	*existential type*

New subtyping rules ($\Gamma \vdash S <: T$)

$$\frac{\Gamma, X<:U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X<:U, S_2\} <: \{\exists X<:U, T_2\}} \quad \text{(S-SOME)}$$

New typing rules ($\Gamma \vdash t : T$)

$$\frac{\Gamma \vdash t_2 : \{X \mapsto U\}T_2 \qquad \Gamma \vdash U <: T_1}{\Gamma \vdash \{\exists X=U, t_2\} \text{ as } \{\exists X <: T_1, T_2\} : \{\exists X <: T_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \exists X <: T_{11}. T_{12} \qquad \Gamma, X <: T_{11}, x:T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X,x\}=t_1 \text{ in } t_2 : T_2} \quad \text{(T-UNPACK)}$$

20.5.1 Exercise: Show how the subtyping rule S-SOME can be obtained from the subtyping rules for universals by extending the encoding of existential types in terms of universal types described in Section 19.3. □

20.6 Historical Notes and Further Reading

The idea of bounded quantification was introduced by Cardelli and Wegner [CW85] in the language Fun. (Their "Kernel Fun" calculus corresponds to our $F^k_{<:}$.) Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [Mit84b], Fun integrated Girard-Reynolds polymorphism [Gir72, Rey74] with Cardelli's first-order calculus of subtyping [**?**, Car84]. The original Fun was simplified and slightly generalized by Bruce and Longo [BL90], and again by Curien and Ghelli [CG92], yielding the calculus we call $F^f_{<:}$.

The most comprehensive single paper on bounded quantification is the survey by Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

Fun and its relatives have been studied extensively by programming language theorists and designers. Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [Car88]. Curien and Ghelli [CG92, Ghe90] address a number of syntactic properties of $F_{<:}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [BL90], Martini [Mar88], Breazu-Tannen, Coquand,

Gunter, and Scedrov [BCGS91], Cardone [Car89], Cardelli and Longo [CL91], Cardelli, Martini, Mitchell, and Scedrov [**?**], Curien and Ghelli [CG92, CG91], and Bruce and Mitchell [BM92]. $F_{<:}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [CM91], Bruce [Bru91], Cardelli [Car92], and Canning, Cook, Hill, Olthoff, and Mitchell [CCH+89]. Bounded quantification also plays a key role in Cardelli's programming language Quest [Car91, CL91] and in the Abel language developed at HP Labs [CCHO89, CCH+89, CHO88, CHC90].

The undecidability of full $F_{<:}$ was shown by Pierce [Pie94] and further analyzed by Ghelli [Ghe95].

The effect of bounded quantification on Church encodings of algebraic datatypes (cf. Section 20.3) was considered by Ghelli thesis [Ghe90] and Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

An extension of $F_{<:}$ with intersection types was studied by Pierce [Pie91a, Pie97] and applied to the modeling of object-oriented languages with multiple inheritance by Compagnoni and Pierce [CP96].

Chapter 21

Implementing Bounded Quantification

Next we consider the problem of building a typechecking algorithm for a language with bounded quantifiers. The algorithm that we construct will be parametric in an algorithm for the subtype relation, which we consider in the following section.

21.1 Promotion

In the typechecking algorithm for $\lambda_{<:}$ in Section 13.2, the key idea was that we can calculate a *minimal* type for each term from the minimal types of its subterms. We will use the same basic idea to typecheck $F^{k}_{<:}$, but we need to take into account one slight complication arising from the presence of type variables in the system.

Consider the term

```
f = λX<:Nat→Nat. λy:X. y 5;
▶ f : ∀X<:Nat→Nat. X → Nat
```

This term is clearly well typed, since the type of the variable `y` in the application `(y 5)` is `X`, which can be promoted to `Nat→Nat` by T-SUB. But the *minimal* type of `y` is not an arrow type. In order to find the minimal type of the application, we need to find the minimal arrow type that `y` possesses—i.e., the minimal arrow type that is a supertype of `X`. Not too surprisingly, the correct way to find this type is to **promote** the minimal type of `y` until it is something other than a type variable.

Formally, write $\Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}$ to mean "`T` is the *least nonvariable supertype* of `S`," defined by repeated promotion of variables as follows:

Exposure

Exposure ($\Gamma \vdash_{\blacktriangleright} T \Uparrow T'$)

$$\frac{X{<:}T \in \Gamma \qquad \Gamma \vdash_{\blacktriangleright} T \Uparrow T'}{\Gamma \vdash_{\blacktriangleright} X \Uparrow T'} \qquad \text{(XA-PROMOTE)}$$

$$\frac{T \text{ is not a type variable}}{\Gamma \vdash_{\blacktriangleright} T \Uparrow T} \qquad \text{(XA-OTHER)}$$

It is easy to check that these rules define a total function. Moreover, the result of promotion is always the least supertype that has some shape other than a variable.

21.1.1 Lemma: Suppose $\Gamma \vdash S \Uparrow T$.

1. $\Gamma \vdash S <: T$.
2. If $\Gamma \vdash S <: U$ and U is not a variable, then $\Gamma \vdash T <: U$. □

Proof: Part (1) is easy. Part (2) goes by straightforward induction on a derivation of $\Gamma \vdash S <: U$. □

21.2 Minimal Typing

The algorithm for calculating minimal types is built along the same basic lines as the one for $\lambda_{<:}$, with one additional twist: the minimal type of a term may always be a type variable, and such a type will need to be promoted to its smallest non-variable supertype (its smallest **concrete** supertype, we might say) in order to be used on the left of an application or type application.

Algorithmic typing

Algorithmic typing ($\Gamma \vdash_{\blacktriangleright} t : T$)

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash_{\blacktriangleright} x : T} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash_{\blacktriangleright} t_2 : T_2}{\Gamma \vdash_{\blacktriangleright} \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t_1 : T_1 \qquad \Gamma \vdash_{\blacktriangleright} T_1 \Uparrow (T_{11} \to T_{12}) \qquad \Gamma \vdash_{\blacktriangleright} t_2 : T_2 \qquad \Gamma \vdash_{\blacktriangleright} T_2 <: T_{11}}{\Gamma \vdash_{\blacktriangleright} t_1\ t_2 : T_{12}} \qquad \text{(TA-APP)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2} \quad \text{(TA-TABS)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash T_1 \Uparrow \forall X <: T_{11} . T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\} T_{12}} \quad \text{(TA-TAPP)}$$

The proofs of soundness and completeness of this algorithm with respect to the original typing rules are fairly routine.

21.2.1 Theorem [Minimal typing]:

1. If $\Gamma \vdash t : T$, then $\Gamma \vdash t : T$.
2. If $\Gamma \vdash t : T$, then $\Gamma \vdash t : M$ where $\Gamma \vdash M <: T$. □

Proof: Part (1) proceeds by a straightforward induction on algorithmic derivations. Part (2) is more interesting; it goes by induction on a derivation of $\Gamma \vdash t : T$. (The most important cases are those for the rules T-APP and T-TAPP.)

Case T-VAR: $t = x \qquad x : T \in \Gamma$

By TA-VAR, $\Gamma \vdash x : T$. By S-REFL, $\Gamma \vdash T <: T$.

Case T-ABS: $t = \lambda x : T_1 . t_2 \qquad \Gamma, x : T_1 \vdash t_2 : T_2 \qquad T = T_1 \to T_2$

By the induction hypothesis, $\Gamma, x : T_1 \vdash t_2 : M_2$ for some M_2 with $\Gamma, x : T_1 \vdash M_2 <: T_2$, i.e. (since subtyping does not depend on term variable bindings), $\Gamma \vdash M_2 <: T_2$. By TA-ABS, $\Gamma \vdash t : T_1 \to M_2$. Finally, by S-REFL and S-ARROW, we have $\Gamma \vdash T_1 \to M_2 <: T_1 \to T_2$.

Case T-APP: $t = t_1\ t_2 \qquad \Gamma \vdash t_1 : T_{11} \to T_{12}$
$T = T_{12} \qquad \Gamma \vdash t_2 : T_{11}$

By the induction hypothesis, we have $\Gamma \vdash t_1 : M_1$ and $\Gamma \vdash t_2 : M_2$, with $\Gamma \vdash M_1 <: T_{11} \to T_{12}$ and $\Gamma \vdash M_2 <: T_{11}$. Let N_1 be the least nonvariable supertype of M_1—i.e., suppose $\Gamma \vdash M_1 \Uparrow N_1$. By the promotion lemma (21.1.1), $\Gamma \vdash N_1 <: T_{11} \to T_{12}$. But we know that N_1 is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $N_1 = N_{11} \to N_{12}$, with $\Gamma \vdash T_{11} <: N_{11}$ and $\Gamma \vdash N_{12} <: T_{12}$. By transitivity, $\Gamma \vdash M_2 <: N_{11}$, so rule TA-APP applies and gives us $\Gamma \vdash t_1\ t_2 : N_{12}$, which satisfies the requirements.

Case T-TABS: $t = \lambda X <: T_1 . t_2 \qquad \Gamma, X <: T_1 \vdash t_2 : T_2 \qquad T = \forall X <: T_1 . T_2$

By the induction hypothesis, $\Gamma, X <: T_1 \vdash t_2 : M_2$ for some M_2 with $\Gamma, X <: T_1 \vdash M_2 <: T_2$. By TA-TABS, $\Gamma \vdash t : \forall X <: T_1 . M_2$. Finally, by S-REFL and S-ALL, we have $\Gamma \vdash \forall X <: T_1 . M_2 <: \forall X <: T_1 . T_2$.

Case T-TAPP: $t = t_1\ [T_2]$ $\quad \Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12}$
$T = \{X \mapsto T_2\}T_{12}$ $\quad \Gamma \vdash T_2 <: T_{11}$

By the induction hypothesis, we have $\Gamma \vdash t_1 : M_1$, with $\Gamma \vdash M_1 <: \forall X{<:}T_{11}.T_{12}$. Let N_1 be the least nonvariable supertype of M_1—i.e., suppose $\Gamma \vdash M_1 \Uparrow N_1$. By the promotion lemma (21.1.1), $\Gamma \vdash N_1 <: \forall X{<:}T_{11}.T_{12}$. But we know that N_1 is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $N_1 = \forall X{<:}N_{11}.N_{12}$, with $N_{11} = T_{11}$ and $\Gamma, X{<:}T_{11} \vdash N_{12} <: T_{12}$. Rule TA-TAPP gives us $\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}N_{12}$, and the preservation of subtyping under substitution (20.4.6) yields $\Gamma \vdash \{X \mapsto T_2\}N_{12} <: \{X \mapsto T_2\}T_{12} = T$.

Case T-SUB: $\Gamma \vdash t : S$ $\quad \Gamma \vdash S <: T$

By the induction hypothesis, $\Gamma \vdash t : M$ with $\Gamma \vdash M <: S$. By S-TRANS, $\Gamma \vdash M <: T$, from which T-SUB yields the desired result. □

21.2.2 Corollary [Decidability of typing]: The $F^k_{<:}$ typing relation is decidable (if we are given a decision procedure for the subtyping relation). □

Proof: Given Γ and t, we can check whether there is some T such that $\Gamma \vdash t : T$ by using the algorithmic typing rules to generate a proof of $\Gamma \vdash t : T$. If we succeed, then this T is also a type for T in the original typing relation (by part (1) of 21.2.1). If not, then part (2) of 21.2.1 implies that t has no type in the original typing relation.

Finally, note that the algorithmic typing rules constitute a terminating algorithm, since they are syntax-directed and always reduce the size of t when read from bottom to top. □

21.2.3 Exercise: Show how to add primitive booleans and conditionals to the minimal typing algorithm for $F^k_{<:}$. (Solution on page 261.) □

21.3 Subtyping in $F^k_{<:}$

As we saw in the simply typed lambda-calculus with subtyping, the subtyping rules in their present form do not constitute an algorithm for deciding the subtyping relation. We cannot use them "from bottom to top," for two reasons:

1. There are some overlaps between the conclusions of different rules (specifically, between S-REFL and nearly all the other rules). That is, looking at the form of a derivable subtyping statement $\Gamma \vdash S <: T$, we cannot decide which of the rules must have been used last in deriving it.

2. More seriously, one rule (S-TRANS) contains a metavariable in the premises that does not appear in the conclusion. To apply this rule from botom to top, we'd need to guess what type to replace this metavariable with.

The overlap between S-REFL and the other rules is easily dealt with, using exactly the same technique as we used in Chapter 13: we remove the full reflexivity rule and replace it by a restricted reflexivity rule that applies only to type variables.

$$\Gamma \vdash X <: X$$

Next we must deal with S-TRANS. Unfortunately, unlike the simple subtyping relation studied in Chapter 13, the transitivity rule here interacts in an important way with another rule—namely S-TVAR, which allows assumptions about type variables to be used in deriving subtyping statements. For example, if

$$\Gamma \quad = \quad W<:\mathtt{Top},\ X<:W,\ Y<:X,\ Z<:Y$$

then the statement

$$\Gamma \vdash Z <: W$$

is provable using all the subtyping rules, but cannot be proved if S-TRANS is removed. That is, an instance of S-TRANS whose left-hand subderivation is an instance of the axiom S-TVAR, as in

$$\dfrac{\dfrac{}{\Gamma \vdash Z <: Y}\,(\text{S-TVAR}) \qquad \dfrac{\vdots}{\Gamma \vdash Y <: W}}{\Gamma \vdash Z <: W}\,(\text{S-TRANS})$$

cannot, in general, be eliminated.

Fortunately, it turns out that derivations of this form are the *only* essential uses of transitivity in subtyping. This observation can be made precise by introducing a new subtyping rule

$$\dfrac{X<:U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

that captures exactly this pattern of variable lookup followed by transitivity, and showing (as we will do below) that replacing the transitivity and variable rules by this one does not change the set of derivable subtyping statements.

These intuitions are summarized in the following definition. The algorithmic subtype relation of $F^k_{<:}$ is the least relation closed under the following rules:

Algorithmic subtyping

Algorithmic subtyping ($\Gamma \vdash\!\!\blacktriangleright S <: T$)

$$\Gamma \vdash\!\!\blacktriangleright S <: \mathtt{Top} \qquad \text{(SA-TOP)}$$

$$\Gamma \vdash\!\!\blacktriangleright X <: X \qquad \text{(SA-REFL-TVAR)}$$

$$\dfrac{X<:U \in \Gamma \qquad \Gamma \vdash\!\!\blacktriangleright U <: T}{\Gamma \vdash\!\!\blacktriangleright X <: T} \qquad \text{(SA-TRANS-TVAR)}$$

$$\frac{\Gamma \vdash\!\!\blacktriangleright T_1 <: S_1 \qquad \Gamma \vdash\!\!\blacktriangleright S_2 <: T_2}{\Gamma \vdash\!\!\blacktriangleright S_1 \to S_2 <: T_1 \to T_2} \quad \text{(SA-ARROW)}$$

$$\frac{\Gamma, X<:U_1 \vdash\!\!\blacktriangleright S_2 <: T_2}{\Gamma \vdash\!\!\blacktriangleright \forall X<:U_1.S_2 <: \forall X<:U_1.T_2} \quad \text{(SA-ALL)}$$

21.3.1 Lemma [Reflexivity of the algorithmic subtype relation]: The statement $\Gamma \vdash\!\!\blacktriangleright T <: T$ is provable for all Γ and T. □

Proof: Easy induction on T. □

21.3.2 Lemma [Transitivity of the algorithmic subtype relation]: If $\Gamma \vdash\!\!\blacktriangleright S <: Q$ and $\Gamma \vdash\!\!\blacktriangleright Q <: T$, then $\Gamma \vdash\!\!\blacktriangleright S <: T$. □

Proof: By induction on the sum of the sizes of the two derivations. Given two derivations of some total size, we proceed by considering the final rules in each.

First, if the right-hand derivation is an instance of SA-TOP, then we are done, since $\Gamma \vdash S <: \mathrm{Top}$ by SA-TOP. Moreover, if the left-hand derivation is an instance of SA-TOP, then $Q = \mathrm{Top}$ and by looking at the algorithmic rules we see that the right-hand derivation must also be an instance of SA-TOP.

If either derivation is an instance of SA-REFL-TVAR, then we are again done since the other derivation is the desired result.

Next, if the left-hand derivation ends with an instance of SA-TRANS-TVAR, then $S = X$ with $X<:U \in \Gamma$ and we have a subderivation with conclusion $\Gamma \vdash U <: Q$. By the induction hypothesis, $\Gamma \vdash\!\!\blacktriangleright U <: T$, and, by SA-TRANS-TVAR again, $\Gamma \vdash\!\!\blacktriangleright X <: T$, as required.

If the left-hand derivation ends with an instance of SA-ARROW, then we have $S = S_1 \to S_2$ and $Q = Q_1 \to Q_2$, with subderivations $\Gamma \vdash\!\!\blacktriangleright Q_1 <: S_1$ and $\Gamma \vdash\!\!\blacktriangleright S_2 <: Q_2$. But, since we have already considered the case where the right-hand derivation is SA-TOP, the only remaining possibility is that the right-hand derivation also ends with SA-ARROW; we therefore have $T = T_1 \to T_2$, and two more subderivations $\Gamma \vdash\!\!\blacktriangleright T_1 <: Q_1$ and $\Gamma \vdash\!\!\blacktriangleright Q_2 <: T_2$. We now apply the induction hypothesis twice, obtaining $\Gamma \vdash\!\!\blacktriangleright T_1 <: S_1$ and $\Gamma \vdash\!\!\blacktriangleright S_2 <: T_2$. Finally, SA-ARROW yields $\Gamma \vdash\!\!\blacktriangleright S_1 \to S_2 <: T_1 \to T_2$, as required.

The case where the left-hand derivation ends with an instance of SA-ALL is similar. □

21.3.3 Theorem [Soundness and completeness of algorithmic subtyping]:

1. If $\Gamma \vdash S <: T$ then $\Gamma \vdash\!\!\blacktriangleright S <: T$.
2. If $\Gamma \vdash\!\!\blacktriangleright S <: T$ then $\Gamma \vdash S <: T$. □

Proof: Both directions proceed by induction on derivations. Soundness is routine. Completeness is also straightforward, since we have already done the hard work (for the reflexivity and transitivity rules of the original subtype relation) in Lemmas 21.3.1 and 21.3.2. □

Finally, we should check that the subtyping rules define an algorithm that is *total*—i.e., that always terminates no matter what input it is given. We do this by assigning a weight to each subtyping statement, and checking that the algorithmic rules all have conclusions with greater weight than their premises.

21.3.4 Definition: The **weight** of a type $\mathtt{T}$ in a context Γ, written $\mathit{weight}_\Gamma(\mathtt{T})$, is defined as follows:

$$\begin{array}{lcll}
\mathit{weight}_\Gamma(\mathtt{X}) & = & \mathit{weight}_{\Gamma_1}(\mathtt{U}) + 1 & \text{if } \Gamma = \Gamma_1, \mathtt{X<:U}, \Gamma_2 \\
\mathit{weight}_\Gamma(\mathtt{Top}) & = & 1 & \\
\mathit{weight}_\Gamma(\mathtt{T_1 \to T_2}) & = & \mathit{weight}_\Gamma(\mathtt{T_1}) + \mathit{weight}_\Gamma(\mathtt{T_2}) + 1 & \\
\mathit{weight}_\Gamma(\forall \mathtt{X<:T_1}.\mathtt{T}) & = & \mathit{weight}_{\Gamma,\mathtt{X<:T_1}}(\mathtt{T_2}) + 1 &
\end{array}$$

The **weight** of a subtyping statement "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$" is the maximum weight of $\mathtt{S}$ and $\mathtt{T}$ in Γ. □

21.3.5 Theorem: The weight of the conclusion in an instance of any of the algorithmic subtyping rules is strictly greater than the weight of any of the premises. □

Proof: Straightforward inspection of the rules. □

21.4 Subtyping in $F^f_{<:}$

The only difference in the full system $F^f_{<:}$ is that the quantifier subtyping rule S-ALL is replaced by the more expressive variant:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \forall \mathtt{X<:S_1}.\mathtt{S_2} <: \forall \mathtt{X<:T_1}.\mathtt{T_2}} \qquad \text{(S-ALL)}$$

The algorithmic subtype relation of $F^f_{<:}$ consists of exactly the same set of rules as the algorithm for $F^k_{<:}$, except that SA-ALL is refined to reflect the new version of S-ALL:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \vdash\!\!\blacktriangleright \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash\!\!\blacktriangleright \forall \mathtt{X<:S_1}.\mathtt{S_2} <: \forall \mathtt{X<:T_1}.\mathtt{T_2}} \qquad \text{(SA-ALL)}$$

As with $F^k_{<:}$, the soundness and completeness of this algorithmic relation with respect to the original subtype relation can be shown easily, once we have established that the algorithmic relation is reflexive and transitive. For reflexivity, the argument is exactly the same as before. For transitivity, on the other hand, the issues are more subtle.