15 *Subtyping*

We have spent the last several chapters studying the typing behavior of a variety of language features within the framework of the simply typed lambda-calculus. This chapter addresses a more fundamental extension: *subtyping* (sometimes called *subtype polymorphism*). Unlike the features we have studied up to now, which could be formulated more or less orthogonally to each other, subtyping is a cross-cutting extension, interacting with most other language features in non-trivial ways.

Subtyping is characteristically found in *object-oriented* languages and is often considered an essential feature of the object-oriented style. We will explore this connection in detail in Chapter 18; for now, though, we present subtyping in a more economical setting with just functions and records, where most of the interesting issues already appear. §15.5 discusses the combination of subtyping with some of the other features we have seen in previous chapters. In the final section (15.6) we consider a more refined semantics for subtyping, in which the use of suptyping corresponds to the insertion of run-time *coercions*.

15.1 Subsumption

Without subtyping, the rules of the simply typed lambda-calculus can be annoyingly rigid. The type system's insistence that argument types exactly match the domain types of functions will lead the typechecker to reject many programs that, to the programmer, seem obviously well-behaved. For example, recall the typing rule for function application:

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

The calculus studied in this chapter is $\lambda_{<:}$, the simply typed lambda-calculus with subtyping (Figure 15-1) and records (15-3); the corresponding OCaml implementation is `rcdsub`. (Some of the examples also use numbers; `fullsub` is needed to check these.)

According to this rule, the well-behaved term

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

is not typable, since the type of the argument is `{x:Nat,y:Nat}`, whereas the function accepts `{x:Nat}`. But, clearly, the function just requires that its argument is a record with a field `x`; it doesn't care what other fields the argument may or may not have. Moreover, we can see this from the type of the function—we don't need to look at its body to verify that it doesn't use any fields besides `x`. It is *always* safe to pass an argument of type `{x:Nat,y:Nat}` to a function that expects type `{x:Nat}`.

The goal of subtyping is to refine the typing rules so that they can accept terms like the one above. We accomplish this by formalizing the intuition that some types are more informative than others: we say that `S` is a *subtype* of `T`, written `S <: T`, to mean that any term of type `S` can safely be used in a context where a term of type `T` is expected. This view of subtyping is often called the *principle of safe substitution*.

A simpler intuition is to read `S <: T` as "every value described by `S` is also described by `T`," that is, "the elements of `S` are a subset of the elements of `T`." We shall see in §15.6 that other, more refined, interpretations of subtyping are sometimes useful, but this *subset semantics* suffices for most purposes.

The bridge between the typing relation and this subtype relation is provided by adding a new typing rule—the so-called rule of *subsumption:*

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

This rule tells us that, if `S <: T`, then every element `t` of `S` is also an element of `T`. For example, if we define the subtype relation so that `{x:Nat,y:Nat} <: {x:Nat}`, then we can use rule T-Sub to derive `⊢ {x=0,y=1} : {x:Nat}`, which is what we need to make our motivating example typecheck.

15.2 The Subtype Relation

The subtype relation is formalized as a collection of inference rules for deriving statements of the form `S <: T`, pronounced "`S` is a subtype of `T`" (or "`T` is a supertype of `S`"). We consider each form of type (function types, record types, etc.) separately; for each one, we introduce one or more rules formalizing situations when it is safe to allow elements of one type of this form to be used where another is expected.

Before we get to the rules for particular type constructors, we make two general stipulations: first, that subtyping should be reflexive,

$$S <: S \tag{S-Refl}$$

and second, that it should be transitive:

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

These rules follow directly from the intuition of safe substitution.

Now, for record types, we have already seen that we want to consider the type $S = \{k_1 : S_1 \ldots k_m : S_m\}$ to be a subtype of $T = \{l_1 : T_1 \ldots l_n : T_n\}$ if T has fewer fields than S. In particular, it is safe to "forget" some fields at the end of a record type. The so-called *width subtyping* rule captures this intuition:

$$\{l_i : T_i \; ^{i \in 1..n+k}\} <: \{l_i : T_i \; ^{i \in 1..n}\} \tag{S-RcdWidth}$$

It may seem surprising that the "smaller" type—the subtype—is the one with *more* fields. The easiest way to understand this is to adopt a more liberal view of record types than we did in §11.8, regarding a record type `{x:Nat}` as describing "the set of all records with *at least* a field `x` of type `Nat`." Values like `{x=3}` and `{x=5}` are elements of this type, and so are values like `{x=3,y=100}` and `{x=3,a=true,b=true}`. Similarly, the record type `{x:Nat, y:Nat}` describes records with *at least* the fields `x` and `y`, both of type `Nat`. Values like `{x=3,y=100}` and `{x=3,y=100,z=true}` are members of this type, but `{x=3}` is not, and neither is `{x=3,a=true,b=true}`. Thus, the set of values belonging to the second type is a proper subset of the set belonging to the first type. A longer record constitutes a more demanding—i.e., more informative—specification, and so describes a smaller set of values.

The width subtyping rule applies only to record types where the common fields are identical. It is also safe to allow the types of individual fields to vary, as long as the types of each corresponding field in the two records are in the subtype relation. The *depth subtyping* rule expresses this intuition:

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i \; ^{i \in 1..n}\} <: \{l_i : T_i \; ^{i \in 1..n}\}} \tag{S-RcdDepth}$$

The following subtyping derivation uses S-RcdWidth and S-RcdDepth together to show that the nested record type `{x:{a:Nat,b:Nat},y:{m:Nat}}` is a subtype of `{x:{a:Nat},y:{}}`:

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\} <: \{a:Nat\}}}\,\text{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\} <: \{\}}}\,\text{S-RcdWidth}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{\}\}}}\,\text{S-RcdDepth}$$

If we want to use S-RcdDepth to refine the type of just a single record field (instead of refining every field, as we did in the example above), we can use S-Refl to obtain trivial subtyping derivations for the other fields.

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\} <: \{a:Nat\}}}\,\text{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\} <: \{m:Nat\}}}\,\text{S-Refl}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{m:Nat\}\}}}\,\text{S-RcdDepth}$$

We can also use the transitivity rule, S-TRANS, to combine width and depth subtyping. For example, we can obtain a supertype by promoting the type of one field while dropping another:

$$\dfrac{\dfrac{}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat,b:Nat\}\}}}\ \text{S-RCDWIDTH} \qquad \dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \text{S-RCDWIDTH}}{\texttt{\{x:\{a:Nat,b:Nat\}\}} <: \texttt{\{x:\{a:Nat\}\}}}\ \text{S-RCDDEPTH}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\}\}}}\ \text{S-TRANS}$$

Our final record subtyping rule arises from the observation that the order of fields in a record does not make any difference to how we can safely use it, since the only thing that we can *do* with records once we've built them—i.e., projecting their fields—is insensitive to the order of fields.

$$\frac{\{k_j : S_{j\ j\in 1..n}\} \text{ is a permutation of } \{l_i : T_{i\ i\in 1..n}\}}{\{k_j : S_{j\ j\in 1..n}\} <: \{l_i : T_{i\ i\in 1..n}\}} \qquad \text{(S-RCDPERM)}$$

For example, S-RCDPERM tells us that `{c:Top,b:Bool,a:Nat}` is a subtype of `{a:Nat,b:Bool,c:Top}`, and vice versa. (This implies that the subtype relation will *not* be anti-symmetric.)

S-RCDPERM can be used in combination with S-RCDWIDTH and S-TRANS to drop fields from anywhere in a record type, not just at the end.

15.2.1 EXERCISE [⋆]: Draw a derivation showing that `{x:Nat,y:Nat,z:Nat}` is a subtype of `{y:Nat}`. □

S-RCDWIDTH, S-RCDDEPTH, and S-RCDPERM each embody a different sort of flexibility in the use of records. For purposes of discussion, it is useful to present them as three separate rules. In particular, there are languages that allow some of them but not others; for example, most variants of Abadi and Cardelli's *object calculus* (1996) omit width subtyping. However, for purposes of implementation it is more convenient to combine them into a single macro-rule that does all three things at once. This rule is discussed in the next chapter (cf. page 211).

Since we are working in a higher-order language, where not only numbers and records but also functions can be passed as arguments to other functions, we must also give a subtyping rule for function types—i.e., we must specify under what circumstances it is safe to use a function of one type in a context where a different function type is expected.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-ARROW)}$$

Notice that the sense of the subtype relation is reversed (*contravariant*) for the argument types in the left-hand premise, while it runs in the same direction (*covariant*) for the result types as for the function types themselves. The intuition is that, if we have a function f of type $S_1 \to S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. The type of f also tells us that it returns elements of type S_2; we can also view these results belonging to any supertype T_2 of S_2. That is, any function f of type $S_1 \to S_2$ can also be viewed as having type $T_1 \to T_2$.

An alternative view is that it is safe to allow a function of one type $S_1 \to S_2$ to be used in a context where another type $T_1 \to T_2$ is expected as long as none of the arguments that may be passed to the function in this context will surprise it (T_1 <: S_1) and none of the results that it returns will surprise the context (S_2 <: T_2).

Finally, it is convenient to have a type that is a supertype of every type. We introduce a new type constant Top, plus a rule that makes Top a maximum element of the subtype relation.

$$S <: \mathsf{Top} \qquad \text{(S-Top)}$$

§15.4 discusses the Top type further.

Formally, the subtype relation is the least relation closed under the rules we have given. For easy reference, Figures 15-1, 15-2, and 15-3 recapitulate the full definition of the simply typed lambda-calculus with records and subtyping, highlighting the syntactic forms and rules we have added in this chapter. Note that the presence of the reflexivity and transitivity rules means that the subtype relation is clearly a *preorder*; however, because of the record permutation rule, it is not a partial order: there are many pairs of distinct types where each is a subtype of the other.

To finish the discussion of the subtype relation, let us verify that the example at the beginning of the chapter now typechecks. Using the following abbreviations to avoid running off the edge of the page,

$$\begin{array}{llll} \mathtt{f} & \stackrel{\text{def}}{=} & \mathtt{\lambda r{:}\{x{:}Nat\}.\ r.x} & \mathtt{Rx} \stackrel{\text{def}}{=} \mathtt{\{x{:}Nat\}} \\ \mathtt{xy} & \stackrel{\text{def}}{=} & \mathtt{\{x=0,y=1\}} & \mathtt{Rxy} \stackrel{\text{def}}{=} \mathtt{\{x{:}Nat,y{:}Nat\}} \end{array}$$

and assuming the usual typing rules for numeric constants, we can construct a derivation for the typing statement $\vdash$ f xy : Nat as follows:

$$\dfrac{\dfrac{\vdots}{\vdash \mathtt{f} : \mathtt{Rx{\to}Nat}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mathtt{0} : \mathtt{Nat}} \quad \dfrac{}{\vdash \mathtt{1} : \mathtt{Nat}}}{\vdash \mathtt{xy} : \mathtt{Rxy}}\,\text{T-Rcd} \qquad \dfrac{}{\mathtt{Rxy} <: \mathtt{Rx}}\,\text{S-RcdWidth}}{\vdash \mathtt{xy} : \mathtt{Rx}}\,\text{T-Sub}}{\vdash \mathtt{f\ xy} : \mathtt{Nat}}\,\text{T-App}$$

→ <: Top — *Based on $\lambda_{\to}$ (9-1)*

Syntax

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
v ::=		*values:*
	λx:T.t	*abstraction value*
T ::=		*types:*
	Top	*maximum type*
	T→T	*type of functions*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

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

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

Subtyping $\boxed{S <: T}$

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

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

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

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

Typing $\boxed{\Gamma \vdash t : T}$

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

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

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

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

Figure 15-1: Simply typed lambda-calculus with subtyping ($\lambda_{<:}$)

15.2.2 EXERCISE [⋆]: Is this the *only* derivation of the statement ⊢ f xy : Nat? □

15.2.3 EXERCISE [⋆]: (1) How many different supertypes does {a:Top,b:Top} have? (2) Can you find an infinite descending chain in the subtype relation—that is, an infinite sequence of types S_0, S_1, etc. such that each S_{i+1} is a subtype of S_i? (3) What about an infinite ascending chain? □

15.2.4 EXERCISE [⋆]: Is there a type that is a subtype of every other type? Is there an arrow type that is a supertype of every other arrow type? □

→ {} *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

t	::=	...	*terms:*
		$\{l_i{=}t_i\ ^{i\in 1..n}\}$	*record*
		$t.l$	*projection*
v	::=	...	*values:*
		$\{l_i{=}v_i\ ^{i\in 1..n}\}$	*record value*
T	::=	...	*types:*
		$\{l_i{:}T_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\{l_i{=}v_i\ ^{i\in 1..n}\}.l_j \longrightarrow v_j \quad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \quad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j, l_k{=}t_k\ ^{k\in j+1..n}\} \longrightarrow \{l_i{=}v_i\ ^{i\in 1..j-1}, l_j{=}t_j', l_k{=}t_k\ ^{k\in j+1..n}\}} \quad \text{(E-Rcd)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i{=}t_i\ ^{i\in 1..n}\} : \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad \text{(T-Proj)}$$

Figure 15-2: Records (same as Figure 11-7)

→ {} <: *Extends* $\lambda_{<:}$ *(15-1) and simple record rules (15-2)*

New subtyping rules $\boxed{S <: T}$

$$\{l_i{:}T_i\ ^{i\in 1..n+k}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\} \quad \text{(S-RcdWidth)}$$

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

$$\frac{\{k_j{:}S_j\ ^{j\in 1..n}\} \text{ is a permutation of } \{l_i{:}T_i\ ^{i\in 1..n}\}}{\{k_j{:}S_j\ ^{j\in 1..n}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\}} \quad \text{(S-RcdPerm)}$$

Figure 15-3: Records and subtyping

15.2.5 Exercise [⋆⋆]: Suppose we extend the calculus with the product type constructor $T_1 \times T_2$ described in §11.6. It is natural to add a subtyping rule

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \quad \text{(S-ProdDepth)}$$

corresponding to S-RcdDepth for records. Would it be a good idea to add a width subtyping rule for products

$$T_1 \times T_2 <: T_1 \quad \text{(S-ProdWidth)}$$

as well? □

15.3 Properties of Subtyping and Typing

Having decided on the definition of the lambda-calculus with subtyping, we now have some work to do to verify that it makes sense—in particular, that the preservation and progress theorems of the simply typed lambda-calculus continue to hold in the presence of subtyping.

15.3.1 EXERCISE [RECOMMENDED, ⋆⋆]: Before reading on, try to predict where difficulties might arise. In particular, suppose we had made a mistake in defining the subtype relation and included a bogus subtyping rule in addition to those above. Which properties of the system can fail? On the other hand, suppose we *omit* one of the subtyping rules—can any properties then break? □

We begin by recording one key property of the subtype relation—an analog of the inversion lemma for the typing relation in the simply typed lambda-calculus (Lemma 9.3.1). If we know that some type S is a subtype of an arrow type, then the subtyping inversion lemma tells us that S itself must be an arrow type; moreover, it tells us that the left-hand sides of the arrows must be (contravariantly) related, and so (covariantly) must the right-hand sides. Similar considerations apply when S is known to be a subtype of a record type: we know that S has more fields (S-RCDWIDTH) in some order (S-RCDPERM), and that the types of common fields are in the subtype relation (S-RCDDEPTH).

15.3.2 LEMMA [INVERSION OF THE SUBTYPE RELATION]:

1. If $S <: T_1 \to T_2$, then S has the form $S_1 \to S_2$, with $T_1 <: S_1$ and $S_2 <: T_2$.
2. If $S <: \{l_i : T_i \ ^{i \in 1..n}\}$, then S has the form $\{k_j : S_j \ ^{j \in 1..m}\}$, with at least the labels $\{l_i \ ^{i \in 1..n}\}$—i.e., $\{l_i \ ^{i \in 1..n}\} \subseteq \{k_j \ ^{j \in 1..m}\}$—and with $S_j <: T_i$ for each common label $l_i = k_j$. □

Proof: EXERCISE [RECOMMENDED, ⋆⋆]. □

To prove that types are preserved during evaluation, we begin with an inversion lemma for the typing relation (cf. Lemma 9.3.1 for the simply typed lambda-calculus). Rather than stating the lemma in its most general form, we give here just the cases that are actually needed in the proof of the preservation theorem below. (The general form can be read off from the algorithmic subtype relation in the next chapter, Definition 16.2.2.)

15.3.3 LEMMA:

1. If $\Gamma \vdash \lambda x : S_1 . \ s_2 : T_1 \to T_2$, then $T_1 <: S_1$ and $\Gamma, x : S_1 \vdash s_2 : T_2$.

2. If $\Gamma \vdash \{k_a{=}s_a\ ^{a\in 1..m}\} : \{l_i{:}T_i\ ^{i\in 1..n}\}$, then $\{l_i\ ^{i\in 1..n}\} \subseteq \{k_a\ ^{a\in 1..m}\}$ and $\Gamma \vdash s_a : T_i$ for each common label $k_a = l_i$. □

Proof: Straightforward induction on typing derivations, using Lemma 15.3.2 for the T-SUB case. □

Next, we need a substitution lemma for the typing relation. The statement of this lemma is unchanged from the simply typed lambda-calculus (Lemma 9.3.8), and its proof is nearly identical.

15.3.4 LEMMA [SUBSTITUTION]: If $\Gamma, x{:}S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$. □

Proof: By induction on typing derivations. We need new cases for T-SUB and for the record typing rules T-RCD and T-PROJ, making straightforward use of the induction hypothesis. The rest is just like the proof of 9.3.8. □

Now, the preservation theorem has the same statement as before. Its proof, though, is somewhat complicated by subtyping at several points.

15.3.5 THEOREM [PRESERVATION]: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$. □

Proof: Straightforward induction on typing derivations. Most of the cases are similar to the proof of preservation for the simply typed lambda-calculus (9.3.9). We need new cases for the record typing rules and for subsumption.

Case T-VAR*:* $t = x$

Can't happen (there are no evaluation rules for variables).

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

Can't happen (t is already a value).

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

From the evaluation rules in Figures 15-1 and 15-2, we see that there are three rules by which $t \longrightarrow t'$ can be derived: E-APP1, E-APP2, and E-APPABS. Proceed by cases.

Subcase E-APP1*:* $t_1 \longrightarrow t_1' \quad t' = t_1'\ t_2$

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

Subcase E-APP2*:* $t_1 = v_1 \quad t_2 \longrightarrow t_2' \quad t' = v_1\ t_2'$

Similar.

Subcase E-APPABS*:* $t_1 = \lambda x{:}S_{11}.\ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12}$

By Lemma 15.3.3(1), $T_{11} <: S_{11}$ and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$. By T-SUB, $\Gamma \vdash t_2 : S_{11}$. From this and the substitution lemma (15.3.4), we obtain $\Gamma \vdash t' : T_{12}$.

Case T-RCD*:* $t = \{l_i{=}t_i\ ^{i\in 1..n}\}$ $\Gamma \vdash t_i : T_i$ for each i
$T = \{l_i{:}T_i\ ^{i\in 1..n}\}$

The only evaluation rule whose left-hand side is a record is E-RCD. From the premise of this rule, we see that $t_j \longrightarrow t'_j$ for some field t_j. The result follows from the induction hypothesis (applied to the corresponding assumption $\Gamma \vdash t_j : T_j$) and T-RCD.

Case T-PROJ*:* $t = t_1.l_j$ $\Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\}$ $T = T_j$

From the evaluation rules in Figures 15-1 and 15-2, we see that there are two rules by which $t \longrightarrow t'$ can be derived: E-PROJ, E-PROJRCD.

Subcase E-PROJ*:* $t_1 \longrightarrow t'_1$ $t' = t'_1.l_j$

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

Subcase E-PROJRCD*:* $t_1 = \{k_a{=}v_a\ ^{a\in 1..m}\}$ $l_j = k_b$ $t' = v_b$

By Lemma 15.3.3(2), we have $\{l_i\ ^{i\in 1..n}\} \subseteq \{k_a\ ^{a\in 1..m}\}$ and $\Gamma \vdash v_a : T_i$ for each $k_a = l_i$. In particular, $\Gamma \vdash v_b : T_j$, as desired.

Case T-SUB*:* $t : S$ $S <: T$

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

To prove that well-typed terms cannot get stuck, we begin (as in Chapter 9) with a canonical forms lemma, which tells us the possible shapes of values belonging to arrow and record types.

15.3.6 LEMMA [CANONICAL FORMS]:

1. If v is a closed value of type $T_1 \to T_2$, then v has the form $\lambda x{:}S_1.t_2$.
2. If v is a closed value of type $\{l_i{:}T_i\ ^{i\in 1..n}\}$, then v has the form $\{k_j{=}v_j\ ^{a\in 1..m}\}$, with $\{l_i\ ^{i\in 1..n}\} \subseteq \{k_a\ ^{a\in 1..m}\}$. □

Proof: EXERCISE [RECOMMENDED, ⋆⋆⋆]. □

The progress theorem and its proof are now quite close to what we saw in the simply typed lambda-calculus. Most of the burden of dealing with subtyping has been pushed into the canonical forms lemma, and only a few small changes are needed here.

15.3.7 THEOREM [PROGRESS]: If t is a closed, well-typed term, then either t is a value or else there is some t' with $t \longrightarrow t'$. □

Proof: By straightforward induction on typing derivations. The variable case cannot occur (because t is closed). The case for lambda-abstractions is immediate, since abstractions are values. The remaining cases are more interesting.

Case T-APP: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2 \quad \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \quad \vdash \mathtt{t}_2 : \mathtt{T}_{11} \quad \mathtt{T} = \mathtt{T}_{12}$

By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation; likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-APP1 applies to $\mathtt{t}$. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-APP2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma (15.3.6) tells us that $\mathtt{t}_1$ has the form $\lambda\mathtt{x}{:}\mathtt{S}_{11}.\mathtt{t}_{12}$, so rule E-APPABS applies to $\mathtt{t}$.

Case T-RCD: $\mathtt{t} = \{\mathtt{l}_i{=}\mathtt{t}_i\ ^{i\in 1..n}\}$ for each $i \in 1..n$, $\vdash \mathtt{t}_i : \mathtt{T}_i$
$\mathtt{T} = \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}$

By the induction hypothesis, each $\mathtt{t}_i$ either is already a value or can make a step of evaluation. If all of them are values, then $\mathtt{t}$ is a value. On the other hand, if at least one can make a step, then rule E-RCD applies to $\mathtt{t}$.

Case T-PROJ: $\mathtt{t} = \mathtt{t}_1.\mathtt{l}_j \quad \vdash \mathtt{t}_1 : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\} \quad \mathtt{T} = \mathtt{T}_j$

By the induction hypothesis, either $\mathtt{t}_1$ is a value or it can make an evaluation step. If $\mathtt{t}_1$ can make a step, then (by E-PROJ) so can $\mathtt{t}$. If $\mathtt{t}_1$ is a value, then by the canonical forms lemma (15.3.6) $\mathtt{t}_1$ has the form $\{\mathtt{k}_a{=}\mathtt{v}_j\ ^{a\in 1..m}\}$, with $\{\mathtt{l}_i\ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_a\ ^{a\in 1..m}\}$ and with $\vdash \mathtt{v}_j : \mathtt{T}_i$ for each $\mathtt{l}_i = \mathtt{k}_j$. In particular, $\mathtt{l}_j$ is among the labels $\{\mathtt{k}_a\ ^{a\in 1..m}\}$ of $\mathtt{t}_1$, from which rule E-PROJRCD tells us that $\mathtt{t}$ itself can take an evaluation step.

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

The result follows directly from the induction hypothesis. □

15.4 The Top and Bottom Types

The maximal type `Top` is not a necessary part of the simply typed lambda-calculus with subtyping; it can be removed without damaging the properties of the system. However, it is included in most presentations, for several reasons. First, it corresponds to the type `Object` found in most object-oriented languages. Second, `Top` is a convenient technical device in more sophisticated systems combining subtyping and parametric polymorphism. For example, in System $F_{<:}$ (Chapters 26 and 28), the presence of `Top` allows us to recover ordinary unbounded quantification from bounded quantification, streamlining the system. Indeed, even records can be encoded in $F_{<:}$, further streamlining the presentation (at least for purposes of formal study); this encoding critically depends on `Top`. Finally, since `Top`'s behavior is straightforward and it is often useful in examples, there is little reason not to keep it.

It is natural to ask whether we can also complete the subtype relation with a *minimal* element—a type `Bot` that is a subtype of every type. The answer is that we can: this extension is formalized in Figure 15-4.

The first thing to notice is that `Bot` is empty—there are no closed values