SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

# SUB

## SUBTYPING

```
Set Warnings "-notation-overridden,-parsing".
Require Import Maps.
Require Import Types.
Require Import Smallstep.
```

# Concepts

We now turn to the study of *subtyping*, a key feature needed to support the object-oriented programming style.

## A Motivating Example

Suppose we are writing a program involving two record types defined as follows:

```
Person  = {name:String, age:Nat}
Student = {name:String, age:Nat, gpa:Nat}
```

In the simply typed lamdba-calculus with records, the term

```
(\r:Person. (r.age)+1) {name="Pat",age=21,gpa=1}
```

is not typable, since it applies a function that wants a two-field record to an argument that actually provides three fields, while the `T_App` rule demands that the domain type of the function being applied must match the type of the argument precisely.

But this is silly: we're passing the function a *better* argument than it needs! The only thing the body of the function can possibly do with its record argument `r` is project the field `age` from it: nothing else is allowed by the type, and the presence or absence of an extra `gpa` field makes no difference at all. So, intuitively, it seems that this function should be applicable to any record value that has at least an `age` field.

More generally, a record with more fields is "at least as good in any context" as one with just a subset of these fields, in the sense that any value belonging to the longer record type can be used *safely* in any context expecting the shorter record type. If the

context expects something with the shorter type but we actually give it something with the longer type, nothing bad will happen (formally, the program will not get stuck).

The principle at work here is called *subtyping*. We say that "`S` is a subtype of `T`", written `S <: T`, if a value of type `S` can safely be used in any context where a value of type `T` is expected. The idea of subtyping applies not only to records, but to all of the type constructors in the language — functions, pairs, etc.

## Subtyping and Object-Oriented Languages

Subtyping plays a fundamental role in many programming languages — in particular, it is closely related to the notion of *subclassing* in object-oriented languages.

An *object* in Java, C#, etc. can be thought of as a record, some of whose fields are functions ("methods") and some of whose fields are data values ("fields" or "instance variables"). Invoking a method `m` of an object `o` on some arguments $a_1..a_n$ roughly consists of projecting out the `m` field of `o` and applying it to $a_1..a_n$.

The type of an object is called a *class* — or, in some languages, an *interface*. It describes which methods and which data fields the object offers. Classes and interfaces are related by the *subclass* and *subinterface* relations. An object belonging to a subclass (or subinterface) is required to provide all the methods and fields of one belonging to a superclass (or superinterface), plus possibly some more.

The fact that an object from a subclass can be used in place of one from a superclass provides a degree of flexibility that is extremely handy for organizing complex libraries. For example, a GUI toolkit like Java's Swing framework might define an abstract interface `Component` that collects together the common fields and methods of all objects having a graphical representation that can be displayed on the screen and interact with the user, such as the buttons, checkboxes, and scrollbars of a typical GUI. A method that relies only on this common interface can now be applied to any of these objects.

Of course, real object-oriented languages include many other features besides these. For example, fields can be updated. Fields and methods can be declared `private`. Classes can give *initializers* that are used when constructing objects. Code in subclasses can cooperate with code in superclasses via *inheritance*. Classes can have static methods and fields. Etc., etc.

To keep things simple here, we won't deal with any of these issues — in fact, we won't even talk any more about objects or classes. (There is a lot of discussion in [Pierce 2002], if you are interested.) Instead, we'll study the core concepts behind the subclass / subinterface relation in the simplified setting of the STLC.

## The Subsumption Rule

Our goal for this chapter is to add subtyping to the simply typed lambda-calculus (with some of the basic extensions from `MoreStlc`). This involves two steps:

- Defining a binary *subtype relation* between types.

- Enriching the typing relation to take subtyping into account.

The second step is actually very simple. We add just a single rule to the typing relation: the so-called *rule of subsumption*:

$$\frac{\texttt{Gamma |- t : S} \qquad \texttt{S <: T}}{\texttt{Gamma |- t : T}} \text{ (T\_Sub)}$$

This rule says, intuitively, that it is OK to "forget" some of what we know about a term.

For example, we may know that `t` is a record with two fields (e.g., `S = {x:A→A, y:B→B}`), but choose to forget about one of the fields (`T = {y:B→B}`) so that we can pass `t` to a function that requires just a single-field record.

## The Subtype Relation

The first step — the definition of the relation `S <: T` — is where all the action is. Let's look at each of the clauses of its definition.

### Structural Rules

To start off, we impose two "structural rules" that are independent of any particular type constructor: a rule of *transitivity*, which says intuitively that, if `S` is better (richer, safer) than `U` and `U` is better than `T`, then `S` is better than `T`...

$$\frac{\texttt{S <: U} \qquad \texttt{U <: T}}{\texttt{S <: T}} \text{ (S\_Trans)}$$

... and a rule of *reflexivity*, since certainly any type `T` is as good as itself:

$$\frac{}{\texttt{T <: T}} \text{ (S\_Refl)}$$

### Products

Now we consider the individual type constructors, one by one, beginning with product types. We consider one pair to be a subtype of another if each of its components is.

$$\frac{\texttt{S}_1 \texttt{ <: T}_1 \qquad \texttt{S}_2 \texttt{ <: T}_2}{\texttt{S}_1 \texttt{ * S}_2 \texttt{ <: T}_1 \texttt{ * T}_2} \text{ (S\_Prod)}$$

### Arrows

The subtyping rule for arrows is a little less intuitive. Suppose we have functions `f` and `g` with these types:

```
f : C → Student
g : (C→Person) → D
```

That is, `f` is a function that yields a record of type `Student`, and `g` is a (higher-order) function that expects its argument to be a function yielding a record of type `Person`.

Also suppose that `Student` is a subtype of `Person`. Then the application `g f` is safe even though their types do not match up precisely, because the only thing `g` can do with `f` is to apply it to some argument (of type `C`); the result will actually be a `Student`, while `g` will be expecting a `Person`, but this is safe because the only thing `g` can then do is to project out the two fields that it knows about (`name` and `age`), and these will certainly be among the fields that are present.

This example suggests that the subtyping rule for arrow types should say that two arrow types are in the subtype relation if their results are:

$$\frac{S_2 <: T_2}{S_1 \to S_2 <: S_1 \to T_2} \text{ (S\_Arrow\_Co)}$$

We can generalize this to allow the arguments of the two arrow types to be in the subtype relation as well:

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

But notice that the argument types are subtypes "the other way round": in order to conclude that $S_1 \to S_2$ to be a subtype of $T_1 \to T_2$, it must be the case that $T_1$ is a subtype of $S_1$. The arrow constructor is said to be *contravariant* in its first argument and *covariant* in its second.

Here is an example that illustrates this:

```
f : Person → C
g : (Student → C) → D
```

The application `g f` is safe, because the only thing the body of `g` can do with `f` is to apply it to some argument of type `Student`. Since `f` requires records having (at least) the fields of a `Person`, this will always work. So `Person → C` is a subtype of `Student → C` since `Student` is a subtype of `Person`.

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 $T_1 \to T_2$.

## Records

What about subtyping for record types?

The basic intuition is that it is always safe to use a "bigger" record in place of a "smaller" one. That is, given a record type, adding extra fields will always result in a subtype. If some code is expecting a record with fields `x` and `y`, it is perfectly safe for it

to receive a record with fields `x`, `y`, and `z`; the `z` field will simply be ignored. For example,

```
{name:String, age:Nat, gpa:Nat} <: {name:String, age:Nat}
{name:String, age:Nat} <: {name:String}
{name:String} <: {}
```

This is known as "width subtyping" for records.

We can also create a subtype of a record type by replacing the type of one of its fields with a subtype. If some code is expecting a record with a field `x` of type `T`, it will be happy with a record having a field `x` of type `S` as long as `S` is a subtype of `T`. For example,

```
{x:Student} <: {x:Person}
```

This is known as "depth subtyping".

Finally, although the fields of a record type are written in a particular order, the order does not really matter. For example,

```
{name:String,age:Nat} <: {age:Nat,name:String}
```

This is known as "permutation subtyping".

We *could* formalize these requirements in a single subtyping rule for records as follows:

$$\frac{\begin{array}{c}\forall\ jk\ \text{in}\ j_1..jn, \\ \exists\ ip\ \text{in}\ i_1..im,\ \text{such that} \\ jk{=}ip\ \text{and}\ Sp <: Tk\end{array}}{\{i_1{:}S_1...im{:}Sm\} <: \{j_1{:}T_1...jn{:}Tn\}}\ \text{(S\_Rcd)}$$

That is, the record on the left should have all the field labels of the one on the right (and possibly more), while the types of the common fields should be in the subtype relation.

However, this rule is rather heavy and hard to read, so it is often decomposed into three simpler rules, which can be combined using `S_Trans` to achieve all the same effects.

First, adding fields to the end of a record type gives a subtype:

$$\frac{n > m}{\{i_1{:}T_1...in{:}Tn\} <: \{i_1{:}T_1...im{:}Tm\}}\ \text{(S\_RcdWidth)}$$

We can use `S_RcdWidth` to drop later fields of a multi-field record while keeping earlier fields, showing for example that `{age:Nat,name:String} <: {name:String}`.

Second, subtyping can be applied inside the components of a compound record type:

$$\frac{S_1 <: T_1 \quad ... \quad Sn <: Tn}{\{i_1{:}S_1...in{:}Sn\} <: \{i_1{:}T_1...in{:}Tn\}}\ \text{(S\_RcdDepth)}$$

For example, we can use `S_RcdDepth` and `S_RcdWidth` together to show that `{y:Student, x:Nat} <: {y:Person}`.

Third, subtyping can reorder fields. For example, we want `{name:String, gpa:Nat, age:Nat} <: Person`. (We haven't quite achieved this yet: using just `S_RcdDepth` and `S_RcdWidth` we can only drop fields from the *end* of a record type.) So we add:

$$\frac{\{i_1:S_1...in:Sn\} \text{ is a permutation of } \{j_1:T_1...jn:Tn\}}{\{i_1:S_1...in:Sn\} <: \{j_1:T_1...jn:Tn\}} \quad \text{(S\_RcdPerm)}$$

It is worth noting that full-blown language designs may choose not to adopt all of these subtyping rules. For example, in Java:

- A subclass may not change the argument or result types of a method of its superclass (i.e., no depth subtyping or no arrow subtyping, depending how you look at it).
- Each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes).
- A class may implement multiple interfaces — so-called "multiple inheritance" of interfaces (i.e., permutation is allowed for interfaces).

**Exercise: 2 stars, recommended (arrow_sub_wrong)**

Suppose we had incorrectly defined subtyping as covariant on both the right and the left of arrow types:

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

Give a concrete example of functions `f` and `g` with the following types...

```
f : Student → Nat
g : (Person → Nat) → Nat
```

... such that the application `g f` will get stuck during execution. (Use informal syntax. No need to prove formally that the application gets stuck.)

□

## Top

Finally, it is convenient to give the subtype relation a maximum element — a type that lies above every other type and is inhabited by all (well-typed) values. We do this by adding to the language one new type constant, called `Top`, together with a subtyping rule that places it above every other type in the subtype relation:

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

The `Top` type is an analog of the `Object` type in Java and C#.

### Summary

In summary, we form the STLC with subtyping by starting with the pure STLC (over some set of base types) and then...

- adding a base type `Top`,
- adding the rule of subsumption

$$\frac{\texttt{Gamma |- t : S} \qquad \texttt{S <: T}}{\texttt{Gamma |- t : T}} \text{ (T\_Sub)}$$

  to the typing relation, and
- defining a subtype relation as follows:

$$\frac{\texttt{S <: U} \qquad \texttt{U <: T}}{\texttt{S <: T}} \text{ (S\_Trans)}$$

$$\frac{}{\texttt{T <: T}} \text{ (S\_Refl)}$$

$$\frac{}{\texttt{S <: Top}} \text{ (S\_Top)}$$

$$\frac{\texttt{S}_1 \texttt{ <: T}_1 \qquad \texttt{S}_2 \texttt{ <: T}_2}{\texttt{S}_1 \texttt{ * S}_2 \texttt{ <: T}_1 \texttt{ * T}_2} \text{ (S\_Prod)}$$

$$\frac{\texttt{T}_1 \texttt{ <: S}_1 \qquad \texttt{S}_2 \texttt{ <: T}_2}{\texttt{S}_1 \texttt{ -> S}_2 \texttt{ <: T}_1 \texttt{ -> T}_2} \text{ (S\_Arrow)}$$

$$\frac{\texttt{n > m}}{\{\texttt{i}_1\texttt{:T}_1\texttt{...in:Tn}\} \texttt{ <: } \{\texttt{i}_1\texttt{:T}_1\texttt{...im:Tm}\}} \text{ (S\_RcdWidth)}$$

$$\frac{\texttt{S}_1 \texttt{ <: T}_1 \quad \texttt{...} \quad \texttt{Sn <: Tn}}{\{\texttt{i}_1\texttt{:S}_1\texttt{...in:Sn}\} \texttt{ <: } \{\texttt{i}_1\texttt{:T}_1\texttt{...in:Tn}\}} \text{ (S\_RcdDepth)}$$

$$\frac{\{\texttt{i}_1\texttt{:S}_1\texttt{...in:Sn}\} \texttt{ is a permutation of } \{\texttt{j}_1\texttt{:T}_1\texttt{...jn:Tn}\}}{\{\texttt{i}_1\texttt{:S}_1\texttt{...in:Sn}\} \texttt{ <: } \{\texttt{j}_1\texttt{:T}_1\texttt{...jn:Tn}\}} \text{ (S\_RcdPerm)}$$

## Exercises

#### Exercise: 1 star, optional (subtype_instances_tf_1)

Suppose we have types `S`, `T`, `U`, and `V` with `S <: T` and `U <: V`. Which of the following subtyping assertions are then true? Write *true* or *false* after each one. (`A`, `B`, and `C` here are base types like `Bool`, `Nat`, etc.)

- `T→S <: T→S`
- `Top→U <: S→Top`
- `(C→C) → (A*B) <: (C→C) → (Top*B)`
- `T→T→U <: S→S→V`
- `(T→T)→U <: (S→S)→V`
- `((T→S)→T)→U <: ((S→T)→S)→V`
- `S*V <: T*U`

□

### Exercise: 2 stars (subtype_order)

The following types happen to form a linear order with respect to subtyping:

- `Top`
- `Top → Student`
- `Student → Person`
- `Student → Top`
- `Person → Student`

Write these types in order from the most specific to the most general.

Where does the type `Top→Top→Student` fit into this order? That is, state how `Top → (Top → Student)` compares with each of the five types above. It may be unrelated to some of them.

□

### Exercise: 1 star (subtype_instances_tf_2)

Which of the following statements are true? Write *true* or *false* after each one.

```
∀ S T,
    S <: T  →
    S→S   <:  T→T

∀ S,
     S <: A→A →
     ∃ T,
        S = T→T  ∧  T <: A

∀ S T1 T2,
     (S <: T1 → T2) →
     ∃ S1 S2,
        S = S1 → S2  ∧  T1 <: S1  ∧  S2 <: T2
```

```
∃ S,
     S <: S→S

∃ S,
     S→S <: S

∀ S T1 T2,
     S <: T1*T2 →
     ∃ S1 S2,
        S = S1*S2  ∧  S1 <: T1  ∧  S2 <: T2
```

□

**Exercise: 1 star (subtype_concepts_tf)**

Which of the following statements are true, and which are false?

- There exists a type that is a supertype of every other type.
- There exists a type that is a subtype of every other type.
- There exists a pair type that is a supertype of every other pair type.
- There exists a pair type that is a subtype of every other pair type.
- There exists an arrow type that is a supertype of every other arrow type.
- There exists an arrow type that is a subtype of every other arrow type.
- There is an infinite descending chain of distinct types in the subtype relation—-that is, an infinite sequence of types $S_0$, $S_1$, etc., such that all the `Si`'s are different and each `S(i+1)` is a subtype of `Si`.
- There is an infinite *ascending* chain of distinct types in the subtype relation—-that is, an infinite sequence of types $S_0$, $S_1$, etc., such that all the `Si`'s are different and each `S(i+1)` is a supertype of `Si`.

□

**Exercise: 2 stars (proper_subtypes)**

Is the following statement true or false? Briefly explain your answer. (Here `TBase n` stands for a base type, where `n` is a string standing for the name of the base type. See the Syntax section below.)

```
∀ T,
     ~(T = TBool ∨ ∃ n, T = TBase n) →
     ∃ S,
        S <: T  ∧  S ≠ T
```

□

**Exercise: 2 stars (small_large_1)**

- What is the *smallest* type `T` ("smallest" in the subtype relation) that makes the following assertion true? (Assume we have `Unit` among the base types and `unit` as a constant of this type.)

```
empty |- (\p:T*Top. p.fst) ((\z:A.z), unit) : A→A
```

- What is the *largest* type `T` that makes the same assertion true?

□

**Exercise: 2 stars (small_large_2)**

- What is the *smallest* type `T` that makes the following assertion true?

```
empty |- (\p:(A→A * B→B). p) ((\z:A.z), (\z:B.z)) : T
```

- What is the *largest* type `T` that makes the same assertion true?

□

**Exercise: 2 stars, optional (small_large_3)**

- What is the *smallest* type `T` that makes the following assertion true?

```
a:A |- (\p:(A*T). (p.snd) (p.fst)) (a , \z:A.z) : A
```

- What is the *largest* type `T` that makes the same assertion true?

□

**Exercise: 2 stars (small_large_4)**

- What is the *smallest* type `T` that makes the following assertion true?

```
∃ S,
  empty |- (\p:(A*T). (p.snd) (p.fst)) : S
```

- What is the *largest* type `T` that makes the same assertion true?

□

**Exercise: 2 stars (smallest_1)**

What is the *smallest* type `T` that makes the following assertion true?

```
∃ S, ∃ t,
  empty |- (\x:T. x x) t : S
```

□

**Exercise: 2 stars (smallest_2)**

What is the *smallest* type `T` that makes the following assertion true?

```
empty |- (\x:Top. x) ((\z:A.z) , (\z:B.z)) : T
```

□

**Exercise: 3 stars, optional (count_supertypes)**

How many supertypes does the record type `{x:A, y:C→C}` have? That is, how many different types `T` are there such that `{x:A, y:C→C} <: T`? (We consider two types to be different if they are written differently, even if each is a subtype of the other. For example, `{x:A,y:B}` and `{y:B,x:A}` are different.)

□

**Exercise: 2 stars (pair_permutation)**

The subtyping rule for product types

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 * S_2 <: T_1 * T_2} \quad \text{(S\_Prod)}$$

intuitively corresponds to the "depth" subtyping rule for records. Extending the analogy, we might consider adding a "permutation" rule

$$\frac{}{T_1 * T_2 <: T_2 * T_1}$$

for products. Is this a good idea? Briefly explain why or why not.

□

# Formal Definitions

Most of the definitions needed to formalize what we've discussed above — in particular, the syntax and operational semantics of the language — are identical to what we saw in the last chapter. We just need to extend the typing relation with the subsumption rule and add a new `Inductive` definition for the subtyping relation. Let's first do the identical bits.

## Core Definitions

### Syntax

In the rest of the chapter, we formalize just base types, booleans, arrow types, `Unit`, and `Top`, omitting record types and leaving product types as an exercise. For the sake of more interesting examples, we'll add an arbitrary set of base types like `String`, `Float`, etc. (Since they are just for examples, we won't bother adding any operations over these base types, but we could easily do so.)

```
Inductive ty : Type :=
  | TTop : ty
  | TBool : ty
  | TBase : string → ty
  | TArrow : ty → ty → ty
  | TUnit : ty
.

Inductive tm : Type :=
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tunit : tm
.
```

## Substitution

The definition of substitution remains exactly the same as for the pure STLC.

```
Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒
      if beq_string x y then s else t
  | tabs y T t1 ⇒
      tabs y T (if beq_string x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒
      tapp (subst x s t1) (subst x s t2)
  | ttrue ⇒
      ttrue
  | tfalse ⇒
      tfalse
  | tif t1 t2 t3 ⇒
      tif (subst x s t1) (subst x s t2) (subst x s t3)
  | tunit ⇒
      tunit
  end.

Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

## Reduction

Likewise the definitions of the `value` property and the `step` relation.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_true :
      value ttrue
  | v_false :
      value tfalse
  | v_unit :
```

```
      value tunit
.

Hint Constructors value.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
        value v2 →
        (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
        t1 ==> t1' →
        (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
        value v1 →
        t2 ==> t2' →
        (tapp v1 t2) ==> (tapp v1 t2')
  | ST_IfTrue : ∀ t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      (tif t1 t2 t3) ==> (tif t1' t2 t3)
where "t1 '==>' t2" := (step t1 t2).

Hint Constructors step.
```

## Subtyping

Now we come to the interesting part. We begin by defining the subtyping relation and developing some of its important technical properties.

The definition of subtyping is just what we sketched in the motivating discussion.

```
Reserved Notation "T '<:' U" (at level 40).

Inductive subtype : ty → ty → Prop :=
  | S_Refl : ∀ T,
      T <: T
  | S_Trans : ∀ S U T,
      S <: U →
      U <: T →
      S <: T
  | S_Top : ∀ S,
      S <: TTop
  | S_Arrow : ∀ S1 S2 T1 T2,
      T1 <: S1 →
      S2 <: T2 →
      (TArrow S1 S2) <: (TArrow T1 T2)
where "T '<:' U" := (subtype T U).
```

Note that we don't need any special rules for base types (`TBool` and `TBase`): they are automatically subtypes of themselves (by `S_Refl`) and `Top` (by `S_Top`), and that's all we want.

```
Hint Constructors subtype.

Module Examples.

Open Scope string_scope.
Notation x := "x".
Notation y := "y".
Notation z := "z".

Notation A := (TBase "A").
Notation B := (TBase "B").
Notation C := (TBase "C").

Notation String := (TBase "String").
Notation Float := (TBase "Float").
Notation Integer := (TBase "Integer").

Example subtyping_example_0 :
  (TArrow C TBool) <: (TArrow C TTop).
  (* C->Bool <: C->Top *)
Proof. auto. Qed.
```

### Exercise: 2 stars, optional (subtyping judgements)

(Wait to do this exercise until after you have added product types to the language — see exercise `products` — at least up to this point in the file).

Recall that, in chapter MoreStlc, the optional section "Encoding Records" describes how records can be encoded as pairs. Using this encoding, define pair types representing the following record types:

```
  Person   := { name : String }
  Student  := { name : String ;
                gpa  : Float }
  Employee := { name : String ;
                ssn  : Integer }

Definition Person : ty
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
Definition Student : ty
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
Definition Employee : ty
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

Now use the definition of the subtype relation to prove the following:

```
Example sub_student_person :
  Student <: Person.
```

```
  Proof.
  (* FILL IN HERE *) Admitted.

  Example sub_employee_person :
    Employee <: Person.
  Proof.
  (* FILL IN HERE *) Admitted.
```
□

The following facts are mostly easy to prove in Coq. To get full benefit from the exercises, make sure you also understand how to prove them on paper!

**Exercise: 1 star, optional (subtyping_example_1)**

```
  Example subtyping_example_1 :
    (TArrow TTop Student) <: (TArrow (TArrow C C) Person).
    (* Top->Student <: (C->C)->Person *)
  Proof with eauto.
    (* FILL IN HERE *) Admitted.
```
□

**Exercise: 1 star, optional (subtyping_example_2)**

```
  Example subtyping_example_2 :
    (TArrow TTop Person) <: (TArrow Person TTop).
    (* Top->Person <: Person->Top *)
  Proof with eauto.
    (* FILL IN HERE *) Admitted.
```
□

```
  End Examples.
```

## Typing

The only change to the typing relation is the addition of the rule of subsumption, `T_Sub`.

```
  Definition context := partial_map ty.

  Reserved Notation "Gamma '|-' t '∈' T" (at level 40).

  Inductive has_type : context → tm → ty → Prop :=
    (* Same as before *)
    | T_Var : ∀ Gamma x T,
        Gamma x = Some T →
        Gamma |- tvar x ∈ T
    | T_Abs : ∀ Gamma x T11 T12 t12,
        Gamma & {{x-->T11}} |- t12 ∈ T12 →
        Gamma |- tabs x T11 t12 ∈ TArrow T11 T12
    | T_App : ∀ T1 T2 Gamma t1 t2,
        Gamma |- t1 ∈ TArrow T1 T2 →
        Gamma |- t2 ∈ T1 →
        Gamma |- tapp t1 t2 ∈ T2
    | T_True : ∀ Gamma,
         Gamma |- ttrue ∈ TBool
    | T_False : ∀ Gamma,
```

```
      Gamma |- tfalse ∈ TBool
  | T_If : ∀ t1 t2 t3 T Gamma,
      Gamma |- t1 ∈ TBool →
      Gamma |- t2 ∈ T →
      Gamma |- t3 ∈ T →
      Gamma |- tif t1 t2 t3 ∈ T
  | T_Unit : ∀ Gamma,
      Gamma |- tunit ∈ TUnit
  (* New rule of subsumption *)
  | T_Sub : ∀ Gamma t S T,
      Gamma |- t ∈ S →
      S <: T →
      Gamma |- t ∈ T

where "Gamma '|-' t '∈' T" := (has_type Gamma t T).

Hint Constructors has_type.
```

The following hints help `auto` and `eauto` construct typing derivations. (See chapter UseAuto for more on hints.)

```
Hint Extern 2 (has_type _ (tapp _ _) _) ⇒
  eapply T_App; auto.
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.

Module Examples2.
Import Examples.
```

Do the following exercises after you have added product types to the language. For each informal typing judgement, write it as a formal statement in Coq and prove it.

**Exercise: 1 star, optional (typing_example_0)**

```
(* empty |- ((\z:A.z), (\z:B.z))
         : (A->A * B->B) *)
(* FILL IN HERE *)
```

□

**Exercise: 2 stars, optional (typing_example_1)**

```
(* empty |- (\x:(Top * B->B). x.snd) ((\z:A.z), (\z:B.z))
         : B->B *)
(* FILL IN HERE *)
```

□

**Exercise: 2 stars, optional (typing_example_2)**

```
(* empty |- (\z:(C->C)->(Top * B->B). (z (\x:C.x)).snd)
              (\z:C->C. ((\z:A.z), (\z:B.z)))
         : B->B *)
(* FILL IN HERE *)
```

□

```
End Examples2.
```

# Properties

The fundamental properties of the system that we want to check are the same as always: progress and preservation. Unlike the extension of the STLC with references (chapter References), we don't need to change the *statements* of these properties to take subtyping into account. However, their proofs do become a little bit more involved.

## Inversion Lemmas for Subtyping

Before we look at the properties of the typing relation, we need to establish a couple of critical structural properties of the subtype relation:

- `Bool` is the only subtype of `Bool`, and
- every subtype of an arrow type is itself an arrow type.

These are called *inversion lemmas* because they play a similar role in proofs as the built-in `inversion` tactic: given a hypothesis that there exists a derivation of some subtyping statement `S <: T` and some constraints on the shape of `S` and/or `T`, each inversion lemma reasons about what this derivation must look like to tell us something further about the shapes of `S` and `T` and the existence of subtype relations between their parts.

**Exercise: 2 stars, optional (sub_inversion_Bool)**

```
Lemma sub_inversion_Bool : ∀ U,
     U <: TBool →
     U = TBool.
+
□
```

**Exercise: 3 stars (sub_inversion_arrow)**

```
Lemma sub_inversion_arrow : ∀ U V₁ V₂,
     U <: TArrow V₁ V₂ →
     ∃ U₁, ∃ U₂,
       U = TArrow U₁ U₂ ∧ V₁ <: U₁ ∧ U₂ <: V₂.
+
□
```

## Canonical Forms

The proof of the progress theorem — that a well-typed non-value can always take a step — doesn't need to change too much: we just need one small refinement. When we're considering the case where the term in question is an application $t_1\ t_2$ where both $t_1$ and $t_2$ are values, we need to know that $t_1$ has the *form* of a lambda-abstraction, so that we can apply the `ST_AppAbs` reduction rule. In the ordinary STLC, this is obvious: we know that $t_1$ has a function type $T_{11}{\to}T_{12}$, and there is only one

rule that can be used to give a function type to a value — rule `T_Abs` — and the form of the conclusion of this rule forces $t_1$ to be an abstraction.

In the STLC with subtyping, this reasoning doesn't quite work because there's another rule that can be used to show that a value has a function type: subsumption. Fortunately, this possibility doesn't change things much: if the last rule used to show `Gamma |- t1 : T11→T12` is subsumption, then there is some *sub*-derivation whose subject is also $t_1$, and we can reason by induction until we finally bottom out at a use of `T_Abs`.

This bit of reasoning is packaged up in the following lemma, which tells us the possible "canonical forms" (i.e., values) of function type.

**Exercise: 3 stars, optional (canonical_forms_of_arrow_types)**

```
Lemma canonical_forms_of_arrow_types : ∀ Gamma s T1 T2,
  Gamma |- s ∈ TArrow T1 T2 →
  value s →
  ∃ x, ∃ S1, ∃ s2,
     s = tabs x S1 s2.
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

Similarly, the canonical forms of type `Bool` are the constants `true` and `false`.

```
Lemma canonical_forms_of_Bool : ∀ Gamma s,
  Gamma |- s ∈ TBool →
  value s →
  s = ttrue ∨ s = tfalse.
Proof with eauto.
```

## Progress

The proof of progress now proceeds just like the one for the pure STLC, except that in several places we invoke canonical forms lemmas...

*Theorem* (Progress): For any term `t` and type `T`, if `empty |- t : T` then `t` is a value or `t ==> t'` for some term `t'`.

*Proof*: Let `t` and `T` be given, with `empty |- t : T`. Proceed by induction on the typing derivation.

The cases for `T_Abs`, `T_Unit`, `T_True` and `T_False` are immediate because abstractions, `unit`, `true`, and `false` are already values. The `T_Var` case is vacuous because variables cannot be typed in the empty context. The remaining cases are more interesting:

- If the last step in the typing derivation uses rule `T_App`, then there are terms $t_1$ $t_2$ and types $T_1$ and $T_2$ such that `t = t1 t2`, `T = T2`, `empty |- t1 : T1 → T2`, and `empty |- t2 : T1`. Moreover, by the induction hypothesis, either $t_1$ is a value or

it steps, and either $t_2$ is a value or it steps. There are three possibilities to consider:

- Suppose $t_1$ `==>` $t_1$`'` for some term $t_1$`'`. Then $t_1$ $t_2$ `==>` $t_1$`'` $t_2$ by `ST_App1`.
- Suppose $t_1$ is a value and $t_2$ `==>` $t_2$`'` for some term $t_2$`'`. Then $t_1$ $t_2$ `==>` $t_1$ $t_2$`'` by rule `ST_App2` because $t_1$ is a value.
- Finally, suppose $t_1$ and $t_2$ are both values. By the canonical forms lemma for arrow types, we know that $t_1$ has the form `\x:`$S_1$`.s2` for some `x`, $S_1$, and $s_2$. But then `(\x:`$S_1$`.s2)` $t_2$ `==>` `[x:=`$t_2$`]`$s_2$ by `ST_AppAbs`, since $t_2$ is a value.

- If the final step of the derivation uses rule `T_If`, then there are terms $t_1$, $t_2$, and $t_3$ such that `t` = `if` $t_1$ `then` $t_2$ `else` $t_3$, with `empty |-` $t_1$ `: Bool` and with `empty |-` $t_2$ `: T` and `empty |-` $t_3$ `: T`. Moreover, by the induction hypothesis, either $t_1$ is a value or it steps.
  - If $t_1$ is a value, then by the canonical forms lemma for booleans, either $t_1$ = `true` or $t_1$ = `false`. In either case, `t` can step, using rule `ST_IfTrue` or `ST_IfFalse`.
  - If $t_1$ can step, then so can `t`, by rule `ST_If`.
- If the final step of the derivation is by `T_Sub`, then there is a type `S` such that `S <: T` and `empty |- t : S`. The desired result is exactly the induction hypothesis for the typing subderivation.

Formally:

```
Theorem progress : ∀ t T,
     empty |- t ∈ T →
     value t ∨ ∃ t', t ==> t'.
+
```

## Inversion Lemmas for Typing

The proof of the preservation theorem also becomes a little more complex with the addition of subtyping. The reason is that, as with the "inversion lemmas for subtyping" above, there are a number of facts about the typing relation that are immediate from the definition in the pure STLC (formally: that can be obtained directly from the `inversion` tactic) but that require real proofs in the presence of subtyping because there are multiple ways to derive the same `has_type` statement.

The following inversion lemma tells us that, if we have a derivation of some typing statement `Gamma |-` `\x:`$S_1$`.t2 : T` whose subject is an abstraction, then there must be some subderivation giving a type to the body $t_2$.

*Lemma*: If `Gamma |- \x:S1.t2 : T`, then there is a type `S2` such that `Gamma & {{x-->S1}} |- t2 : S2` and `S1 → S2 <: T`.

(Notice that the lemma does *not* say, "then `T` itself is an arrow type" — this is tempting, but false!)

*Proof*: Let `Gamma`, `x`, `S1`, `t2` and `T` be given as described. Proceed by induction on the derivation of `Gamma |- \x:S1.t2 : T`. Cases `T_Var`, `T_App`, are vacuous as those rules cannot be used to give a type to a syntactic abstraction.

- If the last step of the derivation is a use of `T_Abs` then there is a type `T12` such that `T = S1 → T12` and `Gamma, x:S1 |- t2 : T12`. Picking `T12` for `S2` gives us what we need: `S1 → T12 <: S1 → T12` follows from `S_Refl`.
- If the last step of the derivation is a use of `T_Sub` then there is a type `S` such that `S <: T` and `Gamma |- \x:S1.t2 : S`. The IH for the typing subderivation tell us that there is some type `S2` with `S1 → S2 <: S` and `Gamma, x:S1 |- t2 : S2`. Picking type `S2` gives us what we need, since `S1 → S2 <: T` then follows by `S_Trans`.

Formally:

```
Lemma typing_inversion_abs : ∀ Gamma x S1 t2 T,
     Gamma |- (tabs x S1 t2) ∈ T →
     ∃ S2,
       TArrow S1 S2 <: T
       ∧ Gamma & {{x-->S1}} |- t2 ∈ S2.
```

+

Similarly...

```
Lemma typing_inversion_var : ∀ Gamma x T,
  Gamma |- (tvar x) ∈ T →
  ∃ S,
    Gamma x = Some S ∧ S <: T.
```

+

```
Lemma typing_inversion_app : ∀ Gamma t1 t2 T2,
  Gamma |- (tapp t1 t2) ∈ T2 →
  ∃ T1,
    Gamma |- t1 ∈ (TArrow T1 T2) ∧
    Gamma |- t2 ∈ T1.
```

+

```
Lemma typing_inversion_true : ∀ Gamma T,
  Gamma |- ttrue ∈ T →
  TBool <: T.
```

+

```
Lemma typing_inversion_false : ∀ Gamma T,
  Gamma |- tfalse ∈ T →
  TBool <: T.
+

Lemma typing_inversion_if : ∀ Gamma t1 t2 t3 T,
  Gamma |- (tif t1 t2 t3) ∈ T →
  Gamma |- t1 ∈ TBool
  ∧ Gamma |- t2 ∈ T
  ∧ Gamma |- t3 ∈ T.
+

Lemma typing_inversion_unit : ∀ Gamma T,
  Gamma |- tunit ∈ T →
    TUnit <: T.
+
```

The inversion lemmas for typing and for subtyping between arrow types can be packaged up as a useful "combination lemma" telling us exactly what we'll actually require below.

```
Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  empty |- (tabs x S1 s2) ∈ (TArrow T1 T2) →
     T1 <: S1
  ∧ empty & {{x-->S1}} |- s2 ∈ T2.
+
```

## Context Invariance

The context invariance lemma follows the same pattern as in the pure STLC.

```
Inductive appears_free_in : string → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
        y ≠ x →
        appears_free_in x t12 →
        appears_free_in x (tabs y T11 t12)
  | afi_if1 : ∀ x t1 t2 t3,
      appears_free_in x t1 →
      appears_free_in x (tif t1 t2 t3)
  | afi_if2 : ∀ x t1 t2 t3,
      appears_free_in x t2 →
      appears_free_in x (tif t1 t2 t3)
  | afi_if3 : ∀ x t1 t2 t3,
```

```
      appears_free_in x t3 →
      appears_free_in x (tif t1 t2 t3)
.

Hint Constructors appears_free_in.

Lemma context_invariance : ∀ Gamma Gamma' t S,
     Gamma |- t ∈ S →
     (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
     Gamma' |- t ∈ S.
+

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   Gamma |- t ∈ T →
   ∃ T', Gamma x = Some T'.
+
```

## Substitution

The *substitution lemma* is proved along the same lines as for the pure STLC. The only significant change is that there are several places where, instead of the built-in `inversion` tactic, we need to use the inversion lemmas that we proved above to extract structural information from assumptions about the well-typedness of subterms.

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     Gamma & {{x-->U}} |- t ∈ S →
     empty |- v ∈ U →
     Gamma |- [x:=v]t ∈ S.
+
```

## Preservation

The proof of preservation now proceeds pretty much as in earlier chapters, using the substitution lemma at the appropriate point and again using inversion lemmas from above to extract structural information from typing assumptions.

*Theorem* (Preservation): If `t`, `t'` are terms and `T` is a type such that `empty |- t : T` and `t ==> t'`, then `empty |- t' : T`.

*Proof*: Let `t` and `T` be given such that `empty |- t : T`. We proceed by induction on the structure of this typing derivation, leaving `t'` general. The cases `T_Abs`, `T_Unit`, `T_True`, and `T_False` cases are vacuous because abstractions and constants don't step. Case `T_Var` is vacuous as well, since the context is empty.

- If the final step of the derivation is by `T_App`, then there are terms $t_1$ and $t_2$ and types $T_1$ and $T_2$ such that `t` = $t_1$ $t_2$, `T` = $T_2$, `empty |-` $t_1$ `:` $T_1$ `→` $T_2$, and `empty |-` $t_2$ `:` $T_1$.

  By the definition of the step relation, there are three ways $t_1$ $t_2$ can step. Cases `ST_App1` and `ST_App2` follow immediately by the induction hypotheses for the

typing subderivations and a use of T_App.

Suppose instead $t_1$ $t_2$ steps by ST_AppAbs. Then $t_1$ = \x:S.t12 for some type S and term $t_{12}$, and t' = [x:=$t_2$]$t_{12}$.

By lemma abs_arrow, we have $T_1$ <: S and x:$S_1$ |- $s_2$ : $T_2$. It then follows by the substitution lemma (substitution_preserves_typing) that empty |- [x:=$t_2$] $t_{12}$ : $T_2$ as desired.

- If the final step of the derivation uses rule T_If, then there are terms $t_1$, $t_2$, and $t_3$ such that t = if $t_1$ then $t_2$ else $t_3$, with empty |- $t_1$ : Bool and with empty |- $t_2$ : T and empty |- $t_3$ : T. Moreover, by the induction hypothesis, if $t_1$ steps to $t_1$' then empty |- $t_1$' : Bool. There are three cases to consider, depending on which rule was used to show t ==> t'.
  - If t ==> t' by rule ST_If, then t' = if $t_1$' then $t_2$ else $t_3$ with $t_1$ ==> $t_1$'. By the induction hypothesis, empty |- $t_1$' : Bool, and so empty |- t' : T by T_If.
  - If t ==> t' by rule ST_IfTrue or ST_IfFalse, then either t' = $t_2$ or t' = $t_3$, and empty |- t' : T follows by assumption.

- If the final step of the derivation is by T_Sub, then there is a type S such that S <: T and empty |- t : S. The result is immediate by the induction hypothesis for the typing subderivation and an application of T_Sub. □

```
Theorem preservation : ∀ t t' T,
     empty |- t ∈ T →
     t ==> t' →
     empty |- t' ∈ T.
```
+

## Records, via Products and Top

This formalization of the STLC with subtyping omits record types for brevity. If we want to deal with them more seriously, we have two choices.

First, we can treat them as part of the core language, writing down proper syntax, typing, and subtyping rules for them. Chapter RecordSub shows how this extension works.

On the other hand, if we are treating them as a derived form that is desugared in the parser, then we shouldn't need any new rules: we should just check that the existing rules for subtyping product and Unit types give rise to reasonable rules for record subtyping via this encoding. To do this, we just need to make one small change to the encoding described earlier: instead of using Unit as the base case in the encoding of tuples and the "don't care" placeholder in the encoding of records, we use Top. So:

```
{a:Nat, b:Nat} ----> {Nat,Nat}       i.e., (Nat,(Nat,Top))
{c:Nat, a:Nat} ----> {Nat,Top,Nat}   i.e., (Nat,(Top,(Nat,Top)))
```

The encoding of record values doesn't change at all. It is easy (and instructive) to check that the subtyping rules above are validated by the encoding.

## Exercises

### Exercise: 2 stars (variations)

Each part of this problem suggests a different way of changing the definition of the STLC with Unit and subtyping. (These changes are not cumulative: each part starts from the original language.) In each part, list which properties (Progress, Preservation, both, or neither) become false. If a property becomes false, give a counterexample.

- Suppose we add the following typing rule:

$$\frac{\texttt{Gamma |- t : } S_1\texttt{->}S_2 \qquad S_1 \texttt{ <: } T_1 \qquad T_1 \texttt{ <: } S_1 \qquad S_2 \texttt{ <: } T_2}{\texttt{Gamma |- t : } T_1\texttt{->}T_2} \text{ (T\_Funny1)}$$

- Suppose we add the following reduction rule:

$$\frac{}{\texttt{unit ==> (\x:Top. x)}} \text{ (ST\_Funny21)}$$

- Suppose we add the following subtyping rule:

$$\frac{}{\texttt{Unit <: Top->Top}} \text{ (S\_Funny3)}$$

- Suppose we add the following subtyping rule:

$$\frac{}{\texttt{Top->Top <: Unit}} \text{ (S\_Funny4)}$$

- Suppose we add the following reduction rule:

$$\frac{}{\texttt{(unit t) ==> (t unit)}} \text{ (ST\_Funny5)}$$

- Suppose we add the same reduction rule *and* a new typing rule:

$$\frac{}{\texttt{(unit t) ==> (t unit)}} \text{ (ST\_Funny5)}$$

$$\frac{}{\texttt{empty |- unit : Top->Top}} \text{ (T\_Funny6)}$$

- Suppose we *change* the arrow subtyping rule to:

$$\frac{S_1 \texttt{ <: } T_1 \quad S_2 \texttt{ <: } T_2}{} \text{ (S\_Arrow')}$$

$$S_1 \to S_2 <: T_1 \to T_2$$

□

# Exercise: Adding Products

**Exercise: 4 stars (products)**

Adding pairs, projections, and product types to the system we have defined is a relatively straightforward matter. Carry out this extension:

- Below, we've added constructors for pairs, first and second projections, and product types to the definitions of `ty` and `tm`.
- Copy the definitions of the substitution function and value relation from above and extend them as in chapter MoreSTLC to include products.
- Similarly, copy and extend the operational semantics with the same reduction rules as in chapter MoreSTLC.
- (Copy and) extend the subtyping relation with this rule:

$$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 * S_2 <: T_1 * T_2} \text{ (Sub\_Prod)}$$

- Extend the typing relation with the same rules for pairs and projections as in chapter MoreSTLC.
- Extend the proofs of progress, preservation, and all their supporting lemmas to deal with the new constructs. (You'll also need to add a couple of completely new lemmas.)

```
Module ProductExtension.

Inductive ty : Type :=
  | TTop : ty
  | TBool : ty
  | TBase : string → ty
  | TArrow : ty → ty → ty
  | TUnit : ty
  | TProd : ty → ty → ty.

Inductive tm : Type :=
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tunit : tm
  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm.
```

Copy and extend and/or fill in required definitions and lemmas here.

```
  Theorem progress : ∀ t T,
       empty |- t ∈ T →
       value t ∨ ∃ t', t ==> t'.
  Proof.
    (* FILL IN HERE *) Admitted.

  Theorem preservation : ∀ t t' T,
       empty |- t ∈ T →
       t ==> t' →
       empty |- t' ∈ T.
  Proof.
    (* FILL IN HERE *) Admitted.

  End ProductExtension.
```

□