STLC

THE SIMPLY TYPED LAMBDA-CALCULUS

The simply typed lambda-calculus (STLC) is a tiny core calculus embodying the key concept of *functional abstraction*, which shows up in pretty much every real-world programming language in some form (functions, procedures, methods, etc.).

We will follow exactly the same pattern as in the previous chapter when formalizing this calculus (syntax, small-step semantics, typing rules) and its main properties (progress and preservation). The new technical challenges arise from the mechanisms of *variable binding* and *substitution*. It which will take some work to deal with these.

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

Overview

The STLC is built on some collection of *base types*: booleans, numbers, strings, etc. The exact choice of base types doesn't matter much — the construction of the language and its theoretical properties work out the same no matter what we choose — so for the sake of brevity let's take just `Bool` for the moment. At the end of the chapter we'll see how to add more base types, and in later chapters we'll enrich the pure STLC with other useful constructs like pairs, records, subtyping, and mutable state.

Starting from boolean constants and conditionals, we add three things:

- variables
- function abstractions
- application

This gives us the following collection of abstract syntax constructors (written out first in informal BNF notation — we'll formalize it below).

```
t ::= x                         variable
    | \x:T1.t2                  abstraction
    | t1 t2                     application
    | true                      constant true
    | false                     constant false
    | if t1 then t2 else t3     conditional
```

The `\` symbol in a function abstraction `\x:T1.t2` is generally written as a Greek letter "lambda" (hence the name of the calculus). The variable `x` is called the *parameter* to the function; the term `t2` is its *body*. The annotation `:T1` specifies the type of arguments that the function can be applied to.

Some examples:

- `\x:Bool. x`

  The identity function for booleans.

- `(\x:Bool. x) true`

  The identity function for booleans, applied to the boolean `true`.

- `\x:Bool. if x then false else true`

  The boolean "not" function.

- `\x:Bool. true`

  The constant function that takes every (boolean) argument to `true`.

- `\x:Bool. \y:Bool. x`

  A two-argument function that takes two booleans and returns the first one. (As in Coq, a two-argument function is really a one-argument function whose body is also a one-argument function.)

- `(\x:Bool. \y:Bool. x) false true`

  A two-argument function that takes two booleans and returns the first one, applied to the booleans `false` and `true`.

  As in Coq, application associates to the left — i.e., this expression is parsed as `((\x:Bool. \y:Bool. x) false) true`.

- `\f:Bool→Bool. f (f true)`

  A higher-order function that takes a *function* `f` (from booleans to booleans) as an argument, applies `f` to `true`, and applies `f` again to the result.

- `(\f:Bool→Bool. f (f true)) (\x:Bool. false)`

  The same higher-order function, applied to the constantly `false` function.

As the last several examples show, the STLC is a language of *higher-order* functions: we can write down functions that take other functions as arguments and/or return other functions as results.

The STLC doesn't provide any primitive syntax for defining *named* functions — all functions are "anonymous." We'll see in chapter `MoreStlc` that it is easy to add named functions to what we've got — indeed, the fundamental naming and binding mechanisms are exactly the same.

The *types* of the STLC include `Bool`, which classifies the boolean constants `true` and `false` as well as more complex computations that yield booleans, plus *arrow types* that classify functions.

```
T ::= Bool
    | T1 → T2
```

For example:

- `\x:Bool. false` has type `Bool→Bool`
- `\x:Bool. x` has type `Bool→Bool`
- `(\x:Bool. x) true` has type `Bool`
- `\x:Bool. \y:Bool. x` has type `Bool→Bool→Bool` (i.e., `Bool → (Bool→Bool)`)
- `(\x:Bool. \y:Bool. x) false` has type `Bool→Bool`
- `(\x:Bool. \y:Bool. x) false true` has type `Bool`

Syntax

We next formalize the syntax of the STLC.

```
Module STLC.
```

Types

```
Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty → ty → ty.
```

Terms

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

Note that an abstraction `\x:T.t` (formally, `tabs x T t`) is always annotated with the type `T` of its parameter, in contrast to Coq (and other functional languages like ML, Haskell, etc.), which use type inference to fill in missing annotations. We're not considering type inference here.

```
Open Scope string_scope.
```

Some examples...

```
Definition x := "x".
Definition y := "y".
Definition z := "z".

Hint Unfold x.
Hint Unfold y.
Hint Unfold z.
```

`idB = \x:Bool. x`

```
Notation idB :=
  (tabs x TBool (tvar x)).
```

`idBB = \x:Bool→Bool. x`

```
Notation idBB :=
  (tabs x (TArrow TBool TBool) (tvar x)).
```

`idBBBB = \x:(Bool→Bool) → (Bool→Bool). x`

```
Notation idBBBB :=
  (tabs x (TArrow (TArrow TBool TBool)
                      (TArrow TBool TBool))
    (tvar x)).
```

`k = \x:Bool. \y:Bool. x`

```
Notation k := (tabs x TBool (tabs y TBool (tvar x))).
```

`notB = \x:Bool. if x then false else true`

```
Notation notB := (tabs x TBool (tif (tvar x) tfalse ttrue)).
```

(We write these as `Notations` rather than `Definitions` to make things easier for `auto`.)

Operational Semantics

To define the small-step semantics of STLC terms, we begin, as always, by defining the set of values. Next, we define the critical notions of *free variables* and *substitution*, which are used in the reduction rule for application expressions. And finally we give the small-step relation itself.

Values

To define the values of the STLC, we have a few cases to consider.

First, for the boolean part of the language, the situation is clear: `true` and `false` are the only values. An `if` expression is never a value.

Second, an application is clearly not a value: It represents a function being invoked on some argument, which clearly still has work left to do.

Third, for abstractions, we have a choice:

- We can say that `\x:T.` t_1 is a value only when t_1 is a value — i.e., only if the function's body has been reduced (as much as it can be without knowing what argument it is going to be applied to).
- Or we can say that `\x:T.` t_1 is always a value, no matter whether t_1 is one or not — in other words, we can say that reduction stops at abstractions.

Our usual way of evaluating expressions in Coq makes the first choice — for example,

```
Compute (fun x:bool ⇒ 3 + 4)
```

yields `fun x:bool ⇒ 7`.

Most real-world functional programming languages make the second choice — reduction of a function's body only begins when the function is actually applied to an argument. We also make the second choice here.

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

Hint Constructors value.
```

Finally, we must consider what constitutes a *complete* program.

Intuitively, a "complete program" must not refer to any undefined variables. We'll see shortly how to define the *free* variables in a STLC term. A complete program is *closed* — that is, it contains no free variables.

(Conversely, a term with free variables is often called an *open term*.)

Having made the choice not to reduce under abstractions, we don't need to worry about whether variables are values, since we'll always be reducing programs "from the outside in," and that means the `step` relation will always be working with closed terms.

Substitution

Now we come to the heart of the STLC: the operation of substituting one term for a variable in another term. This operation is used below to define the operational semantics of function application, where we will need to substitute the argument term for the function parameter in the function's body. For example, we reduce

```
(\x:Bool. if x then true else x) false
```

to

```
if false then true else false
```

by substituting `false` for the parameter `x` in the body of the function.

In general, we need to be able to substitute some given term `s` for occurrences of some variable `x` in another term `t`. In informal discussions, this is usually written `[x:=s]t` and pronounced "substitute `x` with `s` in `t`."

Here are some examples:

- `[x:=true] (if x then x else false)` yields `if true then true else false`
- `[x:=true] x` yields `true`
- `[x:=true] (if x then x else y)` yields `if true then true else y`
- `[x:=true] y` yields `y`
- `[x:=true] false` yields `false` (vacuous substitution)
- `[x:=true] (\y:Bool. if y then x else false)` yields `\y:Bool. if y then true else false`
- `[x:=true] (\y:Bool. x)` yields `\y:Bool. true`
- `[x:=true] (\y:Bool. y)` yields `\y:Bool. y`
- `[x:=true] (\x:Bool. x)` yields `\x:Bool. x`

The last example is very important: substituting `x` with `true` in `\x:Bool. x` does *not* yield `\x:Bool. true`! The reason for this is that the `x` in the body of `\x:Bool. x` is *bound* by the abstraction: it is a new, local name that just happens to be spelled the same as some global name `x`.

Here is the definition, informally...

$$
\begin{array}{lll}
[x:=s]x & = s & \\
[x:=s]y & = y & \text{if } x \neq y \\
[x:=s](\backslash x{:}T_{11}.\ t_{12}) & = \backslash x{:}T_{11}.\ t_{12} & \\
[x:=s](\backslash y{:}T_{11}.\ t_{12}) & = \backslash y{:}T_{11}.\ [x:=s]t_{12} & \text{if } x \neq y \\
[x:=s](t_1\ t_2) & = ([x:=s]t_1)\ ([x:=s]t_2) & \\
[x:=s]\text{true} & = \text{true} & \\
[x:=s]\text{false} & = \text{false} &
\end{array}
$$

```
[x:=s](if t₁ then t₂ else t₃) =
                  if [x:=s]t₁ then [x:=s]t₂ else [x:=s]t₃
```

... and formally:

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

Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' ⇒
      if beq_string x x' then s else t
  | tabs x' T t₁ ⇒
      tabs x' T (if beq_string x x' then t₁ else ([x:=s] t₁))
  | tapp t₁ t₂ ⇒
      tapp ([x:=s] t₁) ([x:=s] t₂)
  | ttrue ⇒
      ttrue
  | tfalse ⇒
      tfalse
  | tif t₁ t₂ t₃ ⇒
      tif ([x:=s] t₁) ([x:=s] t₂) ([x:=s] t₃)
  end

where "'[' x ':=' s ']' t" := (subst x s t).
```

Technical note: Substitution becomes trickier to define if we consider the case where `s`, the term being substituted for a variable in some other term, may itself contain free variables. Since we are only interested here in defining the `step` relation on *closed* terms (i.e., terms like `\x:Bool. x` that include binders for all of the variables they mention), we can avoid this extra complexity here, but it must be dealt with when formalizing richer languages.

For example, using the definition of substitution above to substitute the *open* term `s = \x:Bool. r`, where `r` is a *free* reference to some global resource, for the variable `z` in the term `t = \r:Bool. z`, where `r` is a bound variable, we would get `\r:Bool. \x:Bool. r`, where the free reference to `r` in `s` has been "captured" by the binder at the beginning of `t`.

Why would this be bad? Because it violates the principle that the names of bound variables do not matter. For example, if we rename the bound variable in `t`, e.g., let `t' = \w:Bool. z`, then `[x:=s]t'` is `\w:Bool. \x:Bool. r`, which does not behave the same as `[x:=s]t = \r:Bool. \x:Bool. r`. That is, renaming a bound variable changes how `t` behaves under substitution.

See, for example, [Aydemir 2008] for further discussion of this issue.

Exercise: 3 stars (substi_correct)

The definition that we gave above uses Coq's `Fixpoint` facility to define substitution as a *function*. Suppose, instead, we wanted to define substitution as an inductive *relation* `substi`. We've begun the definition by providing the `Inductive` header and one of the constructors; your job is to fill in the rest of the constructors and prove that the relation you've defined coincides with the function given above.

```
Inductive substi (s:tm) (x:string) : tm → tm → Prop :=
  | s_var1 :
      substi s x (tvar x) s
  (* FILL IN HERE *)
.

Hint Constructors substi.

Theorem substi_correct : ∀ s x t t',
  [x:=s]t = t' ↔ substi s x t t'.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Reduction

The small-step reduction relation for STLC now follows the same pattern as the ones we have seen before. Intuitively, to reduce a function application, we first reduce its left-hand side (the function) until it becomes an abstraction; then we reduce its right-hand side (the argument) until it is also a value; and finally we substitute the argument for the bound variable in the body of the abstraction. This last rule, written informally as

$$(\backslash x{:}T.t12)\ v_2 \implies [x{:=}v_2]t_{12}$$

is traditionally called "beta-reduction".

$$\frac{\text{value } v_2}{(\backslash x{:}T.t12)\ v_2 \implies [x{:=}v_2]t_{12}} \quad \text{(ST\_AppAbs)}$$

$$\frac{t_1 \implies t_1'}{t_1\ t_2 \implies t_1'\ t_2} \quad \text{(ST\_App1)}$$

$$\frac{\begin{array}{c}\text{value } v_1 \\ t_2 \implies t_2'\end{array}}{v_1\ t_2 \implies v_1\ t_2'} \quad \text{(ST\_App2)}$$

... plus the usual rules for conditionals:

$$\frac{}{(\text{if true then } t_1 \text{ else } t_2) \implies t_1} \quad \text{(ST\_IfTrue)}$$

$$\frac{}{(\text{if false then } t_1 \text{ else } t_2) \implies t_2} \quad \text{(ST\_IfFalse)}$$

$$\frac{t_1 \implies t_1'}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \implies (\text{if } t_1' \text{ then } t_2 \text{ else } t_3)} \quad \text{(ST\_If)}$$

Formally:

```
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.

Notation multistep := (multi step).
Notation "t₁ '==>*' t₂" := (multistep t₁ t₂) (at level 40).
```

Examples

Example:

```
      (\x:Bool→Bool. x) (\x:Bool. x) ==>* \x:Bool. x
```

i.e.,

```
      idBB idB ==>* idB
```

```
Lemma step_example1 :
  (tapp idBB idB) ==>* idB.
Proof.
  eapply multi_step.
    apply ST_AppAbs.
    apply v_abs.
  simpl.
  apply multi_refl. Qed.
```

Example:

```
      (\x:Bool→Bool. x) ((\x:Bool→Bool. x) (\x:Bool. x))
            ==>* \x:Bool. x
```

i.e.,

```
(idBB (idBB idB)) ==>* idB.
```

```
Lemma step_example2 :
  (tapp idBB (tapp idBB idB)) ==>* idB.
Proof.
  eapply multi_step.
    apply ST_App2. auto.
    apply ST_AppAbs. auto.
  eapply multi_step.
    apply ST_AppAbs. simpl. auto.
  simpl. apply multi_refl. Qed.
```

Example:

```
(\x:Bool→Bool. x)
   (\x:Bool. if x then false else true)
   true
      ==>* false
```

i.e.,

```
(idBB notB) ttrue ==>* tfalse.
```

```
Lemma step_example3 :
  tapp (tapp idBB notB) ttrue ==>* tfalse.
Proof.
  eapply multi_step.
    apply ST_App1. apply ST_AppAbs. auto. simpl.
  eapply multi_step.
    apply ST_AppAbs. auto. simpl.
  eapply multi_step.
    apply ST_IfTrue. apply multi_refl. Qed.
```

Example:

```
(\x:Bool → Bool. x)
   ((\x:Bool. if x then false else true) true)
      ==>* false
```

i.e.,

```
idBB (notB ttrue) ==>* tfalse.
```

(Note that this term doesn't actually typecheck; even so, we can ask how it reduces.)

```
Lemma step_example4 :
  tapp idBB (tapp notB ttrue) ==>* tfalse.
Proof.
  eapply multi_step.
    apply ST_App2. auto.
    apply ST_AppAbs. auto. simpl.
  eapply multi_step.
    apply ST_App2. auto.
    apply ST_IfTrue.
```

```
    eapply multi_step.
      apply ST_AppAbs. auto. simpl.
    apply multi_refl. Qed.
```

We can use the `normalize` tactic defined in the Types chapter to simplify these proofs.

```
Lemma step_example1' :
  (tapp idBB idB) ==>* idB.
Proof. normalize. Qed.

Lemma step_example2' :
  (tapp idBB (tapp idBB idB)) ==>* idB.
Proof. normalize. Qed.

Lemma step_example3' :
  tapp (tapp idBB notB) ttrue ==>* tfalse.
Proof. normalize. Qed.

Lemma step_example4' :
  tapp idBB (tapp notB ttrue) ==>* tfalse.
Proof. normalize. Qed.
```

Exercise: 2 stars (step_example5)

Try to do this one both with and without `normalize`.

```
Lemma step_example5 :
       tapp (tapp idBBBB idBB) idB
  ==>* idB.
Proof.
  (* FILL IN HERE *) Admitted.

Lemma step_example5_with_normalize :
       tapp (tapp idBBBB idBB) idB
  ==>* idB.
Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Typing

Next we consider the typing relation of the STLC.

Contexts

Question: What is the type of the term "`x y`"?

Answer: It depends on the types of `x` and `y`!

I.e., in order to assign a type to a term, we need to know what assumptions we should make about the types of its free variables.

This leads us to a three-place *typing judgment*, informally written `Gamma ⊢ t ∈ T`, where `Gamma` is a "typing context" — a mapping from variables to their types.

Following the usual notation for partial maps, we could write `Gamma & {{x:T}}` for "update the partial function `Gamma` to also map `x` to `T`."

```
Definition context := partial_map ty.
```

Typing Relation

$$\frac{\texttt{Gamma x = T}}{\texttt{Gamma |- x} \in \texttt{T}} \quad \text{(T\_Var)}$$

$$\frac{\texttt{Gamma \& \{\{ x --> T}_{11}\texttt{ \}\} |- t}_{12} \in \texttt{T}_{12}}{\texttt{Gamma |- \textbackslash x:T}_{11}\texttt{.t12} \in \texttt{T}_{11}\texttt{->T}_{12}} \quad \text{(T\_Abs)}$$

$$\frac{\begin{array}{c}\texttt{Gamma |- t}_1 \in \texttt{T}_{11}\texttt{->T}_{12} \\ \texttt{Gamma |- t}_2 \in \texttt{T}_{11}\end{array}}{\texttt{Gamma |- t}_1\ \texttt{t}_2 \in \texttt{T}_{12}} \quad \text{(T\_App)}$$

$$\frac{}{\texttt{Gamma |- true} \in \texttt{Bool}} \quad \text{(T\_True)}$$

$$\frac{}{\texttt{Gamma |- false} \in \texttt{Bool}} \quad \text{(T\_False)}$$

$$\frac{\texttt{Gamma |- t}_1 \in \texttt{Bool} \qquad \texttt{Gamma |- t}_2 \in \texttt{T} \qquad \texttt{Gamma |- t}_3 \in \texttt{T}}{\texttt{Gamma |- if t}_1\texttt{ then t}_2\texttt{ else t}_3 \in \texttt{T}} \quad \text{(T\_If)}$$

We can read the three-place relation `Gamma |- t ∈ T` as: "under the assumptions in Gamma, the term `t` has the type `T`."

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

Inductive has_type : context → tm → ty → Prop :=
  | 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 : ∀ T11 T12 Gamma t1 t2,
      Gamma |- t1 ∈ TArrow T11 T12 →
      Gamma |- t2 ∈ T11 →
      Gamma |- tapp t1 t2 ∈ T12
  | 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

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

Hint Constructors has_type.
```

Examples

```
Example typing_example_1 :
  empty |- tabs x TBool (tvar x) ∈ TArrow TBool TBool.
Proof.
  apply T_Abs. apply T_Var. reflexivity. Qed.
```

Note that since we added the `has_type` constructors to the hints database, auto can actually solve this one immediately.

```
Example typing_example_1' :
  empty |- tabs x TBool (tvar x) ∈ TArrow TBool TBool.
Proof. auto. Qed.
```

Another example:

```
       empty |- \x:A. \y:A→A. y (y x)
             ∈ A → (A→A) → A.
```

+

Exercise: 2 stars, optional (typing_example_2_full)

Prove the same result without using `auto`, `eauto`, or `eapply` (or `...`).

```
Example typing_example_2_full :
  empty |-
    (tabs x TBool
       (tabs y (TArrow TBool TBool)
          (tapp (tvar y) (tapp (tvar y) (tvar x))))) ∈
    (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (typing_example_3)

Formally prove the following typing derivation holds:

```
       empty |- \x:Bool→B. \y:Bool→Bool. \z:Bool.
                   y (x z)
             ∈ T.
```

```
Example typing_example_3 :
  ∃ T,
    empty |-
      (tabs x (TArrow TBool TBool)
         (tabs y (TArrow TBool TBool)
```

```
               (tabs z TBool
                  (tapp (tvar y) (tapp (tvar x) (tvar z)))))) ∈
         T.
Proof with auto.
  (* FILL IN HERE *) Admitted.
```

☐

We can also show that terms are *not* typable. For example, let's formally check that there is no typing derivation assigning a type to the term `\x:Bool. \y:Bool, x y` — i.e.,

```
¬ ∃ T,
    empty |- \x:Bool. \y:Bool, x y ∈ T.
```

+

Exercise: 3 stars, optional (typing_nonexample_3)

Another nonexample:

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

```
Example typing_nonexample_3 :
  ¬ (∃ S, ∃ T,
        empty |-
          (tabs x S
             (tapp (tvar x) (tvar x))) ∈
          T).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

```
End STLC.
```