SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

STLCPROP

PROPERTIES OF STLC

```
Set Warnings "-notation-overridden,-parsing".
Require Import Maps.
Require Import Types.
Require Import Stlc.
Require Import Smallstep.
Module STLCProp.
Import STLC.
```

In this chapter, we develop the fundamental theory of the Simply Typed Lambda Calculus — in particular, the type safety theorem.

Canonical Forms

As we saw for the simple calculus in the Types chapter, the first step in establishing basic properties of reduction and types is to identify the possible *canonical forms* (i.e., well-typed closed values) belonging to each type. For `Bool`, these are the boolean values `ttrue` and `tfalse`; for arrow types, they are lambda-abstractions.

```
Lemma canonical_forms_bool : ∀ t,
  empty |- t ∈ TBool →
  value t →
  (t = ttrue) ∨ (t = tfalse).
+

Lemma canonical_forms_fun : ∀ t T1 T2,
  empty |- t ∈ (TArrow T1 T2) →
  value t →
  ∃ x u, t = tabs x T1 u.
+
```

Progress

The *progress* theorem tells us that closed, well-typed terms are not stuck: either a well-typed term is a value, or it can take a reduction step. The proof is a relatively straightforward extension of the progress proof we saw in the Types chapter. We'll give the proof in English first, then the formal version.

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

Proof: By induction on the derivation of `|- t ∈ T`.

- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_True`, `T_False`, and `T_Abs` cases are trivial, since in each of these cases we can see by inspecting the rule that `t` is a value.
- If the last rule of the derivation is `T_App`, then `t` has the form $t_1\ t_2$ for some t_1 and t_2, where $\vert\text{-}\ t_1 \in T_2 \to T$ and $\vert\text{-}\ t_2 \in T_2$ for some type T_2. By the induction hypothesis, either t_1 is a value or it can take a reduction step.
  - If t_1 is a value, then consider t_2, which by the other induction hypothesis must also either be a value or take a step.
    - Suppose t_2 is a value. Since t_1 is a value with an arrow type, it must be a lambda abstraction; hence $t_1\ t_2$ can take a step by `ST_AppAbs`.
    - Otherwise, t_2 can take a step, and hence so can $t_1\ t_2$ by `ST_App2`.
  - If t_1 can take a step, then so can $t_1\ t_2$ by `ST_App1`.
- If the last rule of the derivation is `T_If`, then `t` = `if` t_1 `then` t_2 `else` t_3, where t_1 has type `Bool`. By the IH, t_1 either is a value or takes a step.
  - If t_1 is a value, then since it has type `Bool` it must be either `true` or `false`. If it is `true`, then `t` steps to t_2; otherwise it steps to t_3.
  - Otherwise, t_1 takes a step, and therefore so does `t` (by `ST_If`).

+

Exercise: 3 stars, advanced (progress_from_term_ind)

Show that progress can also be proved by induction on terms instead of induction on typing derivations.

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

```
    induction t; intros T Ht; auto.
    (* FILL IN HERE *) Admitted.
```

☐

Preservation

The other half of the type soundness property is the preservation of types during reduction. For this part, we'll need to develop some technical machinery for reasoning about variables and substitution. Working from top to bottom (from the high-level property we are actually interested in to the lowest-level technical lemmas that are needed by various cases of the more interesting proofs), the story goes like this:

- The *preservation theorem* is proved by induction on a typing derivation, pretty much as we did in the Types chapter. The one case that is significantly different is the one for the `ST_AppAbs` rule, whose definition uses the substitution operation. To see that this step preserves typing, we need to know that the substitution itself does. So we prove a...
- *substitution lemma*, stating that substituting a (closed) term `s` for a variable `x` in a term `t` preserves the type of `t`. The proof goes by induction on the form of `t` and requires looking at all the different cases in the definition of substitition. This time, the tricky cases are the ones for variables and for function abstractions. In both, we discover that we need to take a term `s` that has been shown to be well-typed in some context `Gamma` and consider the same term `s` in a slightly different context `Gamma'`. For this we prove a...
- *context invariance* lemma, showing that typing is preserved under "inessential changes" to the context `Gamma` — in particular, changes that do not affect any of the free variables of the term. And finally, for this, we need a careful definition of...
- the *free variables* in a term — i.e., variables that are used in the term and where these uses are *not* in the scope of an enclosing function abstraction binding a variable of the same name.

To make Coq happy, we need to formalize the story in the opposite order...

Free Occurrences

A variable `x` *appears free in* a term *t* if `t` contains some occurrence of `x` that is not under an abstraction labeled `x`. For example:

- `y` appears free, but `x` does not, in `\x:T→U. x y`
- both `x` and `y` appear free in `(\x:T→U. x y) x`
- no variables appear free in `\x:T→U. \y:T. x y`

Formally:

```
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.
```

The *free variables* of a term are just the variables that appear free in it. A term with no free variables is said to be *closed*.

```
Definition closed (t:tm) :=
  ∀ x, ¬ appears_free_in x t.
```

An *open* term is one that may contain free variables. (I.e., every term is an open term; the closed terms are a subset of the open ones. "Open" really means "possibly containing free variables.")

Exercise: 1 star (afi)

In the space below, write out the rules of the `appears_free_in` relation in informal inference-rule notation. (Use whatever notational conventions you like — the point of the exercise is just for you to think a bit about the meaning of each rule.) Although this is a rather low-level, technical definition, understanding it is crucial to understanding substitution and its properties, which are really the crux of the lambda-calculus.

```
(* FILL IN HERE *)
```

□

Substitution

To prove that substitution preserves typing, we first need a technical lemma connecting free variables and typing contexts: If a variable `x` appears free in a term `t`, and if we know `t` is well typed in context `Gamma`, then it must be the case that `Gamma` assigns a type to `x`.

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

Proof: We show, by induction on the proof that `x` appears free in `t`, that, for all contexts `Gamma`, if `t` is well typed under `Gamma`, then `Gamma` assigns some type to `x`.

- If the last rule used is `afi_var`, then `t = x`, and from the assumption that `t` is well typed under `Gamma` we have immediately that `Gamma` assigns a type to `x`.
- If the last rule used is `afi_app1`, then $t = t_1\ t_2$ and `x` appears free in t_1. Since `t` is well typed under `Gamma`, we can see from the typing rules that t_1 must also be, and the IH then tells us that `Gamma` assigns `x` a type.
- Almost all the other cases are similar: `x` appears free in a subterm of `t`, and since `t` is well typed under `Gamma`, we know the subterm of `t` in which `x` appears is well typed under `Gamma` as well, and the IH gives us exactly the conclusion we want.
- The only remaining case is `afi_abs`. In this case $t = \backslash y{:}T_{11}.t12$ and `x` appears free in t_{12}, and we also know that `x` is different from `y`. The difference from the previous cases is that, whereas `t` is well typed under `Gamma`, its body t_{12} is well typed under $(Gamma \& \{\{y{-}{-}{>}T_{11}\}\}$, so the IH allows us to conclude that `x` is assigned some type by the extended context $(Gamma \& \{\{y{-}{-}{>}T_{11}\}\}$. To conclude that `Gamma` assigns a type to `x`, we appeal to lemma `update_neq`, noting that `x` and `y` are different variables.

\+

Next, we'll need the fact that any term `t` that is well typed in the empty context is closed (it has no free variables).

Exercise: 2 stars, optional (typable_empty__closed)

```
Corollary typable_empty__closed : ∀ t T,
    empty |- t ∈ T →
    closed t.
```

\+

☐

Sometimes, when we have a proof `Gamma |- t : T`, we will need to replace `Gamma` by a different context `Gamma'`. When is it safe to do this? Intuitively, it must at least be the case that `Gamma'` assigns the same types as `Gamma` to all the variables that appear free in `t`. In fact, this is the only condition that is needed.

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

Proof: By induction on the derivation of `Gamma |- t ∈ T`.

- If the last rule in the derivation was `T_Var`, then `t = x` and `Gamma x = T`. By assumption, `Gamma' x = T` as well, and hence `Gamma' |- t ∈ T` by `T_Var`.
- If the last rule was `T_Abs`, then $t = \backslash y{:}T_{11}.\ t_{12}$, with $T = T_{11} \rightarrow T_{12}$ and $Gamma\ \&\ \{\{y \text{-->} T_{11}\}\}\ |\text{-}\ t_{12} \in T_{12}$. The induction hypothesis is that, for any context `Gamma''`, if $Gamma\ \&\ \{\{y \text{-->} T_{11}\}\}$ and `Gamma''` assign the same types to all the free variables in t_{12}, then t_{12} has type T_{12} under `Gamma''`. Let `Gamma'` be a context which agrees with `Gamma` on the free variables in `t`; we must show $Gamma'\ |\text{-}\ \backslash y{:}T_{11}.\ t_{12} \in T_{11} \rightarrow T_{12}$.

  By `T_Abs`, it suffices to show that $Gamma'\ \&\ \{\{y \text{-->} T_{11}\}\}\ |\text{-}\ t_{12} \in T_{12}$. By the IH (setting $Gamma'' = Gamma'\ \&\ \{\{y{:}T_{11}\}\}$), it suffices to show that $Gamma\ \&\ \{\{y \text{-->} T_{11}\}\}$ and $Gamma'\ \&\ \{\{y \text{-->} T_{11}\}\}$ agree on all the variables that appear free in t_{12}.

  Any variable occurring free in t_{12} must be either `y` or some other variable. $Gamma\ \&\ \{\{y \text{-->} T_{11}\}\}$ and $Gamma'\ \&\ \{\{y \text{-->} T_{11}\}\}$ clearly agree on `y`. Otherwise, note that any variable other than `y` that occurs free in t_{12} also occurs free in $t = \backslash y{:}T_{11}.\ t_{12}$, and by assumption `Gamma` and `Gamma'` agree on all such variables; hence so do $Gamma\ \&\ \{\{y \text{-->} T_{11}\}\}$ and $Gamma'\ \&\ \{\{y \text{-->} T_{11}\}\}$.
- If the last rule was `T_App`, then $t = t_1\ t_2$, with $Gamma\ |\text{-}\ t_1 \in T_2 \rightarrow T$ and $Gamma\ |\text{-}\ t_2 \in T_2$. One induction hypothesis states that for all contexts `Gamma'`, if `Gamma'` agrees with `Gamma` on the free variables in t_1, then t_1 has type $T_2 \rightarrow T$ under `Gamma'`; there is a similar IH for t_2. We must show that $t_1\ t_2$ also has type `T` under `Gamma'`, given the assumption that `Gamma'` agrees with `Gamma` on all the free variables in $t_1\ t_2$. By `T_App`, it suffices to show that t_1 and t_2 each have the same type under `Gamma'` as under `Gamma`. But all free variables in t_1 are also free in $t_1\ t_2$, and similarly for t_2; hence the desired result follows from the induction hypotheses.

+

Now we come to the conceptual heart of the proof that reduction preserves types — namely, the observation that *substitution* preserves types.

Formally, the so-called *substitution lemma* says this: Suppose we have a term `t` with a free variable `x`, and suppose we've assigned a type `T` to `t` under the assumption that `x` has some type U. Also, suppose that we have some other term v and that we've shown that v has type U. Then, since v satisfies the assumption we made about x when typing t, we can substitute v for each of the occurrences of x in t and obtain a new term that still has type T.

Lemma: If Gamma & {{x-->U}} |- t ∈ T and |- v ∈ U, then Gamma |- [x:=v]t ∈ T.

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

One technical subtlety in the statement of the lemma is that we assume v has type U in the *empty* context — in other words, we assume v is closed. This assumption considerably simplifies the T_Abs case of the proof (compared to assuming Gamma |- v ∈ U, which would be the other reasonable assumption at this point) because the context invariance lemma then tells us that v has type U in any context at all — we don't have to worry about free variables in v clashing with the variable being introduced into the context by T_Abs.

The substitution lemma can be viewed as a kind of commutation property. Intuitively, it says that substitution and typing can be done in either order: we can either assign types to the terms t and v separately (under suitable contexts) and then combine them using substitution, or we can substitute first and then assign a type to [x:=v] t — the result is the same either way.

Proof: We show, by induction on t, that for all T and Gamma, if Gamma & {{x-->U}} |- t ∈ T and |- v ∈ U, then Gamma |- [x:=v]t ∈ T.

- If t is a variable there are two cases to consider, depending on whether t is x or some other variable.
  - If t = x, then from the fact that Gamma & {{x-->U}} |- x ∈ T we conclude that U = T. We must show that [x:=v]x = v has type T under Gamma, given the assumption that v has type U = T under the empty context. This follows from context invariance: if a closed term has type T in the empty context, it has that type in any context.
  - If t is some variable y that is not equal to x, then we need only note that y has the same type under Gamma & {{x-->U}} as under Gamma.
- If t is an abstraction \y:T_{11}. t_{12}, then the IH tells us, for all Gamma' and T', that if Gamma' & {{x-->U}} |- t_{12} ∈ T' and |- v ∈ U, then Gamma' |- [x:=v]t_{12} ∈ T'.

  The substitution in the conclusion behaves differently depending on whether x and y are the same variable.

First, suppose x = y. Then, by the definition of substitution, [x:=v]t = t, so we just need to show Gamma |- t ∈ T. But we know Gamma & {{x-->U}} |- t : T, and, since y does not appear free in \y:T_{11}. t_{12}, the context invariance lemma yields Gamma |- t ∈ T.

Second, suppose x ≠ y. We know Gamma & {{x-->U; y-->T_{11}}} |- t_{12} ∈ T_{12} by inversion of the typing relation, from which Gamma & {{y-->T_{11}; x-->U}} |- t_{12} ∈ T_{12} follows by the context invariance lemma, so the IH applies, giving us Gamma & {{y-->T_{11}}} |- [x:=v]t_{12} ∈ T_{12}. By T_Abs, Gamma |- \y:T_{11}. [x:=v]t_{12} ∈ T_{11}→T_{12}, and by the definition of substitution (noting that x ≠ y), Gamma |- \y:T_{11}. [x:=v]t_{12} ∈ T_{11}→T_{12} as required.

- If t is an application t_1 t_2, the result follows straightforwardly from the definition of substitution and the induction hypotheses.
- The remaining cases are similar to the application case.

Technical note: This proof is a rare case where an induction on terms, rather than typing derivations, yields a simpler argument. The reason for this is that the assumption Gamma & {{x-->U}} |- t ∈ T is not completely generic, in the sense that one of the "slots" in the typing relation — namely the context — is not just a variable, and this means that Coq's native induction tactic does not give us the induction hypothesis that we want. It is possible to work around this, but the needed generalization is a little tricky. The term t, on the other hand, is completely generic.

+

Main Theorem

We now have the tools we need to prove preservation: if a closed term t has type T and takes a step to t', then t' is also a closed term with type T. In other words, the small-step reduction relation preserves types.

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

Proof: By induction on the derivation of |- t ∈ T.

- We can immediately rule out T_Var, T_Abs, T_True, and T_False as the final rules in the derivation, since in each of these cases t cannot take a step.
- If the last rule in the derivation is T_App, then t = t_1 t_2. There are three cases to consider, one for each rule that could be used to show that t_1 t_2 takes a step to t'.

  - If t_1 t_2 takes a step by `ST_App1`, with t_1 stepping to t_1', then by the IH t_1' has the same type as t_1, and hence t_1' t_2 has the same type as t_1 t_2.
  - The `ST_App2` case is similar.
  - If t_1 t_2 takes a step by `ST_AppAbs`, then t_1 = `\x:`T_{11}`.t12` and t_1 t_2 steps to `[x:=`t_2`]`t_{12}; the desired result now follows from the fact that substitution preserves types.
- If the last rule in the derivation is `T_If`, then `t` = `if` t_1 `then` t_2 `else` t_3, and there are again three cases depending on how `t` steps.
  - If `t` steps to t_2 or t_3, the result is immediate, since t_2 and t_3 have the same type as `t`.
  - Otherwise, `t` steps by `ST_If`, and the desired conclusion follows directly from the induction hypothesis.

+

Exercise: 2 stars, recommended (subject_expansion_stlc)

An exercise in the Types chapter asked about the *subject expansion* property for the simple language of arithmetic and boolean expressions. Does this property hold for STLC? That is, is it always the case that, if `t ==> t'` and `has_type t' T`, then `empty |- t ∈ T`? If so, prove it. If not, give a counter-example not involving conditionals.

You can state your counterexample informally in words, with a brief explanation.

```
(* FILL IN HERE *)
```

☐

Type Soundness

Exercise: 2 stars, optional (type_soundness)

Put progress and preservation together and show that a well-typed term can *never* reach a stuck state.

```
Definition stuck (t:tm) : Prop :=
  (normal_form step) t ∧ ¬ value t.

Corollary soundness : ∀ t t' T,
  empty |- t ∈ T →
  t ==>* t' →
  ~(stuck t').
```

+

☐

Uniqueness of Types

Exercise: 3 stars (types_unique)

Another nice property of the STLC is that types are unique: a given term (in a given context) has at most one type. Formalize this statement as a theorem called `unique_types`, and prove your theorem.

```
(* FILL IN HERE *)
```

□

Additional Exercises

Exercise: 1 star (progress_preservation_statement)

Without peeking at their statements above, write down the progress and preservation theorems for the simply typed lambda-calculus (as Coq theorems). You can write `Admitted` for the proofs.

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars (stlc_variation1)

Suppose we add a new term `zap` with the following reduction rule

```
                         (ST_Zap)
        ----------------
          t ==> zap
```

and the following typing rule:

```
                              (T_Zap)
        ----------------------
          Gamma |- zap : T
```

Which of the following properties of the STLC remain true in the presence of these rules? For each property, write either "remains true" or "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars (stlc_variation2)

Suppose instead that we add a new term `foo` with the following reduction rules:

$$\frac{}{\texttt{(\textbackslash x:A. x) ==> foo}} \quad \text{(ST\_Foo1)}$$

$$\frac{}{\texttt{foo ==> true}} \quad \text{(ST\_Foo2)}$$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

☐

Exercise: 2 stars (stlc_variation3)

Suppose instead that we remove the rule `ST_App1` from the `step` relation. Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

☐

Exercise: 2 stars, optional (stlc_variation4)

Suppose instead that we add the following new rule to the reduction relation:

$$\frac{}{\texttt{(if true then } t_1 \texttt{ else } t_2\texttt{) ==> true}} \quad \text{(ST\_FunnyIfTrue)}$$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars, optional (stlc_variation5)

Suppose instead that we add the following new rule to the typing relation:

$$\frac{\texttt{Gamma} \mid\!- \; t_1 \in \texttt{Bool->Bool->Bool} \qquad \texttt{Gamma} \mid\!- \; t_2 \in \texttt{Bool}}{\texttt{Gamma} \mid\!- \; t_1 \; t_2 \in \texttt{Bool}} \quad \text{(T\_FunnyApp)}$$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars, optional (stlc_variation6)

Suppose instead that we add the following new rule to the typing relation:

$$\frac{\texttt{Gamma} \mid\!- \; t_1 \in \texttt{Bool} \qquad \texttt{Gamma} \mid\!- \; t_2 \in \texttt{Bool}}{\texttt{Gamma} \mid\!- \; t_1 \; t_2 \in \texttt{Bool}} \quad \text{(T\_FunnyApp')}$$

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars, optional (stlc_variation7)

Suppose we add the following new rule to the typing relation of the STLC:

$$\frac{}{} \quad \text{(T\_FunnyAbs)}$$

```
|- \x:Bool.t ∈ Bool
```

Which of the following properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`

```
(* FILL IN HERE *)
```

- Progress

```
(* FILL IN HERE *)
```

- Preservation

```
(* FILL IN HERE *)
```

□

```
End STLCProp.
```

Exercise: STLC with Arithmetic

To see how the STLC might function as the core of a real programming language, let's extend it with a concrete base type of numbers and some constants and primitive operators.

```
Module STLCArith.
Import STLC.
```

To types, we add a base type of natural numbers (and remove booleans, for brevity).

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

To terms, we add natural number constants, along with successor, predecessor, multiplication, and zero-testing.

```
Inductive tm : Type :=
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm.
```

Exercise: 4 stars (stlc_arith)

Finish formalizing the definition and properties of the STLC extended with arithmetic. This is a longer exercise. Specifically:

1. Copy the core definitions for STLC that we went through, as well as the key lemmas and theorems, and paste them into the file at this point. Do not copy examples,

exercises, etc. (In particular, make sure you don't copy any of the □ comments at the end of exercises, to avoid confusing the autograder.)

You should copy over five definitions:

- Fixpoint susbt
- Inductive value
- Inductive step
- Inductive has_type
- Inductive appears_free_in

And five theorems, with their proofs:

- Lemma context_invariance
- Lemma free_in_context
- Lemma substitution_preserves_typing
- Theorem preservation
- Theorem progress

It will be helpful to also copy over "Reserved Notation", "Notation", and "Hint Constructors" for these things.

2. Edit and extend the five definitions (subst, value, step, has_type, and appears_free_in) so they are appropriate for the new STLC extended with arithmetic.

3. Extend the proofs of all the five properties of the original STLC to deal with the new syntactic forms. Make sure Coq accepts the whole file.

```
(* FILL IN HERE *)
```

□

```
End STLCArith.
```