SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

MORESTLC

MORE ON THE SIMPLY TYPED LAMBDA-CALCULUS

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

Simple Extensions to STLC

The simply typed lambda-calculus has enough structure to make its theoretical properties interesting, but it is not much of a programming language.

In this chapter, we begin to close the gap with real-world languages by introducing a number of familiar features that have straightforward treatments at the level of typing.

Numbers

As we saw in exercise `stlc_arith` at the end of the `StlcProp` chapter, adding types, constants, and primitive operations for natural numbers is easy — basically just a matter of combining the Types and Stlc chapters. Adding more realistic numeric types like machine integers and floats is also straightforward, though of course the specifications of the numeric primitives become more fiddly.

Let Bindings

When writing a complex expression, it is useful to be able to give names to some of its subexpressions to avoid repetition and increase readability. Most languages provide one or more ways of doing this. In OCaml (and Coq), for example, we can write `let x=`t_1 `in` t_2 to mean "reduce the expression t_1 to a value and bind the name `x` to this value while reducing t_2."

Our `let`-binder follows OCaml in choosing a standard *call-by-value* evaluation order, where the `let`-bound term must be fully reduced before reduction of the `let`-body can begin. The typing rule `T_Let` tells us that the type of a `let` can be calculated by calculating the type of the `let`-bound term, extending the context with a binding with this type, and in this enriched context calculating the type of the body (which is then the type of the whole `let` expression).

At this point in the book, it's probably easier simply to look at the rules defining this new feature than to wade through a lot of English text conveying the same information. Here they are:

Syntax:

```
       t ::=                Terms
           | ...                (other terms same as before)
           | let x=t in t       let-binding
```

Reduction:

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{let } x=t_1 \texttt{ in } t_2 \texttt{ ==> let } x=t_1' \texttt{ in } t_2} \quad \text{(ST\_Let1)}$$

$$\frac{}{\texttt{let } x=v_1 \texttt{ in } t_2 \texttt{ ==> } [x:=v_1]t_2} \quad \text{(ST\_LetValue)}$$

Typing:

$$\frac{\text{Gamma} \mid\!- \ t_1 : T_1 \qquad \text{Gamma} \ \& \ \{\{x \dashrightarrow T_1\}\} \mid\!- \ t_2 : T_2}{\text{Gamma} \mid\!- \texttt{ let } x=t_1 \texttt{ in } t_2 : T_2} \quad \text{(T\_Let)}$$

Pairs

Our functional programming examples in Coq have made frequent use of *pairs* of values. The type of such a pair is called a *product type*.

The formalization of pairs is almost too simple to be worth discussing. However, let's look briefly at the various parts of the definition to emphasize the common pattern.

In Coq, the primitive way of extracting the components of a pair is *pattern matching*. An alternative is to take `fst` and `snd` — the first- and second-projection operators — as primitives. Just for fun, let's do our pairs this way. For example, here's how we'd write a function that takes a pair of numbers and returns the pair of their sum and difference:

```
       \x : Nat*Nat.
          let sum = x.fst + x.snd in
          let diff = x.fst - x.snd in
          (sum,diff)
```

Adding pairs to the simply typed lambda-calculus, then, involves adding two new forms of term — pairing, written (t_1,t_2), and projection, written `t.fst` for the first projection from `t` and `t.snd` for the second projection — plus one new type constructor, T_1*T_2, called the *product* of T_1 and T_2.

Syntax:

```
       t ::=                Terms
           | (t,t)              pair
           | t.fst              first projection
           | t.snd              second projection
           | ...

       v ::=                Values
           | (v,v)              pair value
           | ...

       T ::=                Types
           | T * T              product type
           | ...
```

For reduction, we need several new rules specifying how pairs and projection behave.

$$\frac{t_1 \texttt{ ==> } t_1'}{(t_1,t_2) \texttt{ ==> } (t_1',t_2)} \quad \text{(ST\_Pair1)}$$

$$\frac{t_2 \texttt{ ==> } t_2'}{(v_1,t_2) \texttt{ ==> } (v_1,t_2')} \quad \text{(ST\_Pair2)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{t_1.\texttt{fst} \texttt{ ==> } t_1'.\texttt{fst}} \quad \text{(ST\_Fst1)}$$

$$\frac{}{(v_1,v_2).\texttt{fst} \texttt{ ==> } v_1} \quad \text{(ST\_FstPair)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{t_1.\texttt{snd} \texttt{ ==> } t_1'.\texttt{snd}} \quad \text{(ST\_Snd1)}$$

$$\frac{}{(v_1,v_2).\texttt{snd} \texttt{ ==> } v_2} \quad \text{(ST\_SndPair)}$$

Rules `ST_FstPair` and `ST_SndPair` say that, when a fully reduced pair meets a first or second projection, the result is the appropriate component. The congruence rules `ST_Fst1` and `ST_Snd1` allow reduction to proceed under projections, when the term being projected from has not yet been fully reduced. `ST_Pair1` and `ST_Pair2` reduce the parts of pairs: first the left part, and then — when a value appears on the left — the right part. The ordering arising from the use of the metavariables `v` and `t` in these rules enforces a left-to-right evaluation strategy for pairs. (Note the implicit convention that metavariables like `v` and v_1 can only denote values.) We've also added a clause to the definition of values, above, specifying that (v_1,v_2) is a value. The fact that the components of a pair value must themselves be values ensures that a pair passed as an argument to a function will be fully reduced before the function body starts executing.

The typing rules for pairs and projections are straightforward.

$$\frac{\texttt{Gamma |- } t_1 : T_1 \qquad \texttt{Gamma |- } t_2 : T_2}{\texttt{Gamma |- } (t_1,t_2) : T_1{*}T_2} \quad \text{(T\_Pair)}$$

$$\frac{\texttt{Gamma |- } t_1 : T_{11}{*}T_{12}}{\texttt{Gamma |- } t_1\texttt{.fst} : T_{11}} \quad \text{(T\_Fst)}$$

$$\frac{\texttt{Gamma |- } t_1 : T_{11}{*}T_{12}}{\texttt{Gamma |- } t_1\texttt{.snd} : T_{12}} \quad \text{(T\_Snd)}$$

`T_Pair` says that (t_1,t_2) has type T_1*T_2 if t_1 has type T_1 and t_2 has type T_2. Conversely, `T_Fst` and `T_Snd` tell us that, if t_1 has a product type $T_{11}*T_{12}$ (i.e., if it will reduce to a pair), then the types of the projections from this pair are T_{11} and T_{12}.

Unit

Another handy base type, found especially in languages in the ML family, is the singleton type `Unit`. It has a single element — the term constant `unit` (with a small `u`) — and a typing rule making `unit` an element of `Unit`. We also add `unit` to the set of possible values — indeed, `unit` is the *only* possible result of reducing an expression of type `Unit`.

Syntax:

```
       t ::=                Terms
           | unit               unit value
           | ...

       v ::=                Values
           | unit               unit
           | ...

       T ::=                Types
           | Unit               Unit type
           | ...
```

Typing:

$$\frac{}{\texttt{Gamma |- unit : Unit}} \quad \text{(T\_Unit)}$$

It may seem a little strange to bother defining a type that has just one element — after all, wouldn't every computation living in such a type be trivial?

This is a fair question, and indeed in the STLC the `Unit` type is not especially critical (though we'll see two uses for it below). Where `Unit` really comes in handy is in richer languages with *side effects* — e.g., assignment statements that mutate variables or pointers, exceptions and other sorts of nonlocal control structures, etc. In such languages, it is convenient to have a type for the (trivial) result of an expression that is evaluated only for its effect.

Sums

Many programs need to deal with values that can take two distinct forms. For example, we might identify employees in an accounting application using *either* their name *or* their id number. A search function might return *either* a matching value *or* an error code.

These are specific examples of a binary *sum type* (sometimes called a *disjoint union*), which describes a set of values drawn from one of two given types, e.g.:

```
       Nat + Bool
```

We create elements of these types by *tagging* elements of the component types. For example, if `n` is a `Nat` then `inl n` is an element of `Nat+Bool`; similarly, if `b` is a `Bool` then `inr b` is a `Nat+Bool`. The names of the tags `inl` and `inr` arise from thinking of them as functions

```
       inl : Nat -> Nat + Bool
       inr : Bool -> Nat + Bool
```

that "inject" elements of `Nat` or `Bool` into the left and right components of the sum type `Nat+Bool`. (But note that we don't actually treat them as functions in the way we formalize them: `inl` and `inr` are keywords, and `inl t` and `inr t` are primitive syntactic forms, not function applications.)

In general, the elements of a type $T_1 + T_2$ consist of the elements of T_1 tagged with the token `inl`, plus the elements of T_2 tagged with `inr`.

One important usage of sums is signaling errors:

```
      div : Nat -> Nat -> (Nat + Unit) =
      div =
        \x:Nat. \y:Nat.
          if iszero y then
            inr unit
          else
            inl ...
```

The type `Nat + Unit` above is in fact isomorphic to `option nat` in Coq — i.e., it's easy to write functions that translate back and forth.

To *use* elements of sum types, we introduce a `case` construct (a very simplified form of Coq's `match`) to destruct them. For example, the following procedure converts a `Nat+Bool` into a `Nat`:

```
getNat =
  \x:Nat+Bool.
    case x of
      inl n => n
    | inr b => if b then 1 else 0
```

More formally...

Syntax:

```
t ::=                  Terms
    | inl T t              tagging (left)
    | inr T t              tagging (right)
    | case t of            case
        inl x => t
      | inr x => t
    | ...

v ::=                  Values
    | inl T v              tagged value (left)
    | inr T v              tagged value (right)
    | ...

T ::=                  Types
    | T + T                sum type
    | ...
```

Reduction:

$$\frac{\texttt{t}_1 \texttt{ ==> } \texttt{t}_1\texttt{'}}{\texttt{inl T t}_1 \texttt{ ==> inl T t}_1\texttt{'}} \quad \text{(ST\_Inl)}$$

$$\frac{\texttt{t}_1 \texttt{ ==> } \texttt{t}_1\texttt{'}}{\texttt{inr T t}_1 \texttt{ ==> inr T t}_1\texttt{'}} \quad \text{(ST\_Inr)}$$

$$\frac{\texttt{t}_0 \texttt{ ==> } \texttt{t}_0\texttt{'}}{\begin{array}{c}\texttt{case t}_0 \texttt{ of inl x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr x}_2 \Rightarrow \texttt{t}_2 \texttt{ ==>} \\ \texttt{case t}_0\texttt{' of inl x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr x}_2 \Rightarrow \texttt{t}_2\end{array}} \quad \text{(ST\_Case)}$$

$$\frac{}{\begin{array}{c}\texttt{case (inl T v}_0\texttt{) of inl x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr x}_2 \Rightarrow \texttt{t}_2 \\ \texttt{==> [x}_1\texttt{:=v}_0\texttt{]t}_1\end{array}} \quad \text{(ST\_CaseInl)}$$

$$\frac{}{\begin{array}{c}\texttt{case (inr T v}_0\texttt{) of inl x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr x}_2 \Rightarrow \texttt{t}_2 \\ \texttt{==> [x}_2\texttt{:=v}_0\texttt{]t}_2\end{array}} \quad \text{(ST\_CaseInr)}$$

Typing:

$$\frac{\texttt{Gamma |- } t_1 : T_1}{\texttt{Gamma |- inl } T_2\ t_1 : T_1 + T_2} \quad \text{(T\_Inl)}$$

$$\frac{\texttt{Gamma |- } t_1 : T_2}{\texttt{Gamma |- inr } T_1\ t_1 : T_1 + T_2} \quad \text{(T\_Inr)}$$

$$\frac{\begin{array}{c}\texttt{Gamma |- } t_0 : T_1+T_2 \\ \texttt{Gamma , } x_1{:}T_1 \texttt{ |- } t_1 : T \\ \texttt{Gamma , } x_2{:}T_2 \texttt{ |- } t_2 : T\end{array}}{\texttt{Gamma |- case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 : T} \quad \text{(T\_Case)}$$

We use the type annotation in `inl` and `inr` to make the typing relation simpler, similarly to what we did for functions.

Without this extra information, the typing rule `T_Inl`, for example, would have to say that, once we have shown that t_1 is an element of type T_1, we can derive that `inl` t_1 is an element of $T_1 + T_2$ for *any* type T_2. For example, we could derive both `inl 5 : Nat + Nat` and `inl 5 : Nat + Bool` (and infinitely many other types). This peculiarity (technically, a failure of uniqueness of types) would mean that we cannot build a typechecking algorithm simply by "reading the rules from bottom to top" as we could for all the other features seen so far.

There are various ways to deal with this difficulty. One simple one — which we've adopted here — forces the programmer to explicitly annotate the "other side" of a sum type when performing an injection. This is a bit heavy for programmers (so real languages adopt other solutions), but it is easy to understand and formalize.

Lists

The typing features we have seen can be classified into *base types* like `Bool`, and *type constructors* like `→` and `*` that build new types from old ones. Another useful type constructor is `List`. For every type `T`, the type `List T` describes finite-length lists whose elements are drawn from `T`.

In principle, we could encode lists using pairs, sums and *recursive* types. But giving semantics to recursive types is non-trivial. Instead, we'll just discuss the special case of lists directly.

Below we give the syntax, semantics, and typing rules for lists. Except for the fact that explicit type annotations are mandatory on `nil` and cannot appear on `cons`, these lists are essentially identical to those we built in Coq. We use `lcase` to destruct lists, to avoid dealing with questions like "what is the `head` of the empty list?"

For example, here is a function that calculates the sum of the first two elements of a list of numbers:

```
      \x:List Nat.
      lcase x of nil => 0
      | a::x' => lcase x' of nil => a
                     | b::x'' => a+b
```

Syntax:

```
       t ::=                Terms
           | nil T
           | cons t t
           | lcase t of nil => t | x::x => t
           | ...

       v ::=                Values
           | nil T              nil value
           | cons v v           cons value
           | ...

       T ::=                Types
           | List T             list of Ts
           | ...
```

Reduction:

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{cons } t_1\ t_2 \texttt{ ==> cons } t_1'\ t_2} \quad \text{(ST\_Cons1)}$$

$$\frac{t_2 \texttt{ ==> } t_2'}{\texttt{cons } v_1\ t_2 \texttt{ ==> cons } v_1\ t_2'} \quad \text{(ST\_Cons2)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\begin{array}{c}\texttt{(lcase } t_1 \texttt{ of nil} \Rightarrow t_2 \mid \texttt{xh::xt} \Rightarrow t_3\texttt{) ==>} \\ \texttt{(lcase } t_1' \texttt{ of nil} \Rightarrow t_2 \mid \texttt{xh::xt} \Rightarrow t_3\texttt{)}\end{array}} \quad \text{(ST\_Lcase1)}$$

$$\frac{}{\begin{array}{c}\texttt{(lcase nil T of nil} \Rightarrow t_2 \mid \texttt{xh::xt} \Rightarrow t_3\texttt{)} \\ \texttt{==> } t_2\end{array}} \quad \text{(ST\_LcaseNil)}$$

$$\frac{}{\begin{array}{c}\texttt{(lcase (cons vh vt) of nil} \Rightarrow t_2 \mid \texttt{xh::xt} \Rightarrow t_3\texttt{)} \\ \texttt{==> [xh:=vh,xt:=vt]}t_3\end{array}} \quad \text{(ST\_LcaseCons)}$$

Typing:

$$\frac{}{\texttt{Gamma |- nil T : List T}} \quad \text{(T\_Nil)}$$

$$\frac{\texttt{Gamma |- } t_1 \texttt{ : T} \qquad \texttt{Gamma |- } t_2 \texttt{ : List T}}{\texttt{Gamma |- cons } t_1\ t_2\texttt{: List T}} \quad \text{(T\_Cons)}$$

$$\texttt{Gamma |- } t_1 \texttt{ : List } T_1$$

$$\frac{\begin{array}{c}\texttt{Gamma |- } t_2 \texttt{ : T} \\ \texttt{Gamma , h:}T_1\texttt{, t:List }T_1\texttt{ |- } t_3 \texttt{ : T}\end{array}}{\texttt{Gamma |- (lcase } t_1 \texttt{ of nil} \Rightarrow t_2 \texttt{ | h::t} \Rightarrow t_3\texttt{) : T}} \quad \text{(T\_Lcase)}$$

General Recursion

Another facility found in most programming languages (including Coq) is the ability to define recursive functions. For example, we might like to be able to define the factorial function like this:

```
       fact = \x:Nat.
                 if x=0 then 1 else x * (fact (pred x)))
```

Note that the right-hand side of this binder mentions the variable being bound — something that is not allowed by our formalization of `let` above.

Directly formalizing this "recursive definition" mechanism is possible, but it requires a bit of extra effort: in particular, we'd have to pass around an "environment" of recursive function definitions in the definition of the `step` relation.

Here is another way of presenting recursive functions that is equally powerful (though not quite as convenient for the programmer) and more straightforward to formalize: instead of writing recursive definitions, we define a *fixed-point operator* called `fix` that performs the "unfolding" of the recursive definition in the right-hand side as needed, during reduction.

For example, instead of

```
       fact = \x:Nat.
                 if x=0 then 1 else x * (fact (pred x)))
```

we will write:

```
       fact =
           fix
             (\f:Nat->Nat.
                \x:Nat.
                   if x=0 then 1 else x * (f (pred x)))
```

We can derive the latter from the former as follows:

- In the right-hand side of the definition of `fact`, replace recursive references to `fact` by a fresh variable `f`.
- Add an abstraction binding `f` at the front, with an appropriate type annotation. (Since we are using `f` in place of `fact`, which had type `Nat→Nat`, we should require `f` to have the same type.) The new abstraction has type `(Nat→Nat) → (Nat→Nat)`.
- Apply `fix` to this abstraction. This application has type `Nat→Nat`.
- Use all of this as the right-hand side of an ordinary `let`-binding for `fact`.

The intuition is that the higher-order function `f` passed to `fix` is a *generator* for the `fact` function: if `f` is applied to a function that "approximates" the desired behavior of `fact` up to some number `n` (that is, a function that returns correct results on inputs less than or equal to `n` but we don't care what it does on inputs greater than `n`), then `f` returns a slightly better approximation to `fact` — a function that returns correct results for inputs up to `n+1`. Applying `fix` to this generator returns its *fixed point*, which is a function that gives the desired behavior for all inputs `n`.

(The term "fixed point" is used here in exactly the same sense as in ordinary mathematics, where a fixed point of a function `f` is an input `x` such that `f(x) = x`. Here, a fixed point of a function `F` of type `(Nat→Nat)->(Nat→Nat)` is a function `f` of type `Nat→Nat` such that `F f` behaves the same as `f`.)

Syntax:

```
       t ::=                Terms
           | fix t              fixed-point operator
           | ...
```

Reduction:

$$\frac{t_1 \Longrightarrow t_1'}{\texttt{fix } t_1 \Longrightarrow \texttt{fix } t_1'} \quad \text{(ST\_Fix1)}$$

$$\frac{}{\texttt{fix } (\backslash xf{:}T_1.t2) \Longrightarrow [xf{:=}\texttt{fix } (\backslash xf{:}T_1.t2)]\ t_2} \quad \text{(ST\_FixAbs)}$$

Typing:

$$\frac{\text{Gamma} \vdash t_1 : T_1 \to T_1}{\text{Gamma} \vdash \texttt{fix } t_1 : T_1} \quad \text{(T\_Fix)}$$

Let's see how `ST_FixAbs` works by reducing `fact 3 = fix F 3`, where

```
    F = (\f. \x. if x=0 then 1 else x * (f (pred x)))
```

(type annotations are omitted for brevity).

```
    fix F 3
==> ST_FixAbs + ST_App1
    (\x. if x=0 then 1 else x * (fix F (pred x))) 3
==> ST_AppAbs
    if 3=0 then 1 else 3 * (fix F (pred 3))
==> ST_If0_Nonzero
    3 * (fix F (pred 3))
==> ST_FixAbs + ST_Mult2
    3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 3))
==> ST_PredNat + ST_Mult2 + ST_App2
    3 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 2)
==> ST_AppAbs + ST_Mult2
    3 * (if 2=0 then 1 else 2 * (fix F (pred 2)))
==> ST_If0_Nonzero + ST_Mult2
    3 * (2 * (fix F (pred 2)))
```

```
==> ST_FixAbs + 2 x ST_Mult2
    3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 2)))
==> ST_PredNat + 2 x ST_Mult2 + ST_App2
    3 * (2 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 1))
==> ST_AppAbs + 2 x ST_Mult2
    3 * (2 * (if 1=0 then 1 else 1 * (fix F (pred 1))))
==> ST_If0_Nonzero + 2 x ST_Mult2
    3 * (2 * (1 * (fix F (pred 1))))
==> ST_FixAbs + 3 x ST_Mult2
    3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) (pred 1))))
==> ST_PredNat + 3 x ST_Mult2 + ST_App2
    3 * (2 * (1 * ((\x. if x=0 then 1 else x * (fix F (pred x))) 0)))
==> ST_AppAbs + 3 x ST_Mult2
    3 * (2 * (1 * (if 0=0 then 1 else 0 * (fix F (pred 0)))))
==> ST_If0Zero + 3 x ST_Mult2
    3 * (2 * (1 * 1))
==> ST_MultNats + 2 x ST_Mult2
    3 * (2 * 1)
==> ST_MultNats + ST_Mult2
    3 * 2
==> ST_MultNats
    6
```

One important point to note is that, unlike `Fixpoint` definitions in Coq, there is nothing to prevent functions defined using `fix` from diverging.

Exercise: 1 star, optional (halve_fix)

Translate this informal recursive definition into one using `fix`:

```
      halve =
        \x:Nat.
           if x=0 then 0
           else if (pred x)=0 then 0
           else 1 + (halve (pred (pred x))))
(* FILL IN HERE *)
```

□

Exercise: 1 star, optional (fact_steps)

Write down the sequence of steps that the term `fact 1` goes through to reduce to a normal form (assuming the usual reduction rules for arithmetic operations).

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

□

The ability to form the fixed point of a function of type `T→T` for any `T` has some surprising consequences. In particular, it implies that *every* type is inhabited by some term. To see this, observe that, for every type `T`, we can define the term

```
fix (\x:T.x)
```

By `T_Fix` and `T_Abs`, this term has type `T`. By `ST_FixAbs` it reduces to itself, over and over again. Thus it is a *diverging element* of `T`.

More usefully, here's an example using `fix` to define a two-argument recursive function:

```
equal =
  fix
    (\eq:Nat->Nat->Bool.
       \m:Nat. \n:Nat.
         if m=0 then iszero n
         else if n=0 then false
         else eq (pred m) (pred n))
```

And finally, here is an example where `fix` is used to define a *pair* of recursive functions (illustrating the fact that the type T_1 in the rule `T_Fix` need not be a function type):

```
evenodd =
  fix
    (\eo: (Nat->Bool * Nat->Bool).
       let e = \n:Nat. if n=0 then true  else eo.snd (pred n) in
       let o = \n:Nat. if n=0 then false else eo.fst (pred n) in
       (e,o))

even = evenodd.fst
odd  = evenodd.snd
```

Records

As a final example of a basic extension of the STLC, let's look briefly at how to define *records* and their types. Intuitively, records can be obtained from pairs by two straightforward generalizations: they are n-ary (rather than just binary) and their fields are accessed by *label* (rather than position).

Syntax:

```
t ::=                               Terms
    | {i1=t1, ..., in=tn}               record
    | t.i                               projection
    | ...

v ::=                               Values
    | {i1=v1, ..., in=vn}               record value
    | ...
```

```
       T ::=                          Types
           | {i1:T1, ..., in:Tn}        record type
           | ...
```

The generalization from products should be pretty obvious. But it's worth noticing the ways in which what we've actually written is even *more* informal than the informal syntax we've used in previous sections and chapters: we've used "`...`" in several places to mean "any number of these," and we've omitted explicit mention of the usual side condition that the labels of a record should not contain any repetitions.

Reduction:

```
                    ti ==> ti'
           ------------------------------------        (ST_Rcd)
            {i1=v1, ..., im=vm, in=ti, ...}
           ==> {i1=v1, ..., im=vm, in=ti', ...}

                     t1 ==> t1'
                  ----------------                     (ST_Proj1)
                  t1.i ==> t1'.i

               ---------------------------             (ST_ProjRcd)
               {..., i=vi, ...}.i ==> vi
```

Again, these rules are a bit informal. For example, the first rule is intended to be read "if `ti` is the leftmost field that is not a value and if `ti` steps to `ti'`, then the whole record steps..." In the last rule, the intention is that there should only be one field called i, and that all the other fields must contain values.

The typing rules are also simple:

```
        Gamma |- t1 : T1      ...      Gamma |- tn : Tn
    -------------------------------------------------------    (T_Rcd)
    Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}

              Gamma |- t : {..., i:Ti, ...}
              -----------------------------                     (T_Proj)
                     Gamma |- t.i : Ti
```

There are several ways to approach formalizing the above definitions.

- We can directly formalize the syntactic forms and inference rules, staying as close as possible to the form we've given them above. This is conceptually straightforward, and it's probably what we'd want to do if we were building a real compiler (in particular, it will allow us to print error messages in the form that programmers will find easy to understand). But the formal versions of the rules will not be very pretty or easy to work with, because all the `...`s above will have to be replaced with explicit quantifications or comprehensions. For this reason, records are not included in the extended exercise at the end of this chapter. (It is still useful to discuss them informally here because they will help motivate the addition of subtyping to the type system when we get to the Sub chapter.)

- Alternatively, we could look for a smoother way of presenting records — for example, a binary presentation with one constructor for the empty record and another constructor for adding a single field to an existing record, instead of a single monolithic constructor that builds a whole record at once. This is the right way to go if we are primarily interested in studying the metatheory of the calculi with records, since it leads to clean and elegant definitions and proofs. Chapter Records shows how this can be done.
- Finally, if we like, we can avoid formalizing records altogether, by stipulating that record notations are just informal shorthands for more complex expressions involving pairs and product types. We sketch this approach in the next section.

Encoding Records (Optional)

Let's see how records can be encoded using just pairs and `unit`.

First, observe that we can encode arbitrary-size *tuples* using nested pairs and the `unit` value. To avoid overloading the pair notation `(t1,t2)`, we'll use curly braces without labels to write down tuples, so `{}` is the empty tuple, `{5}` is a singleton tuple, `{5,6}` is a 2-tuple (morally the same as a pair), `{5,6,7}` is a triple, etc.

```
{}                 ----> unit
{t1, t2, ..., tn}  ----> (t1, trest)
                          where {t2, ..., tn} ----> trest
```

Similarly, we can encode tuple types using nested product types:

```
{}                 ----> Unit
{T1, T2, ..., Tn}  ----> T1 * TRest
                          where {T2, ..., Tn} ----> TRest
```

The operation of projecting a field from a tuple can be encoded using a sequence of second projections followed by a first projection:

```
t.0        ----> t.fst
t.(n+1)    ----> (t.snd).n
```

Next, suppose that there is some total ordering on record labels, so that we can associate each label with a unique natural number. This number is called the *position* of the label. For example, we might assign positions like this:

LABEL	POSITION
a	0
b	1
c	2
...	...
bar	1395
...	...
foo	4460
...	...

We use these positions to encode record values as tuples (i.e., as nested pairs) by sorting the fields according to their positions. For example:

```
      {a=5, b=6}      --->   {5,6}
      {a=5, c=7}      --->   {5,unit,7}
      {c=7, a=5}      --->   {5,unit,7}
      {c=5, b=3}      --->   {unit,3,5}
      {f=8,c=5,a=7}   --->   {7,unit,5,unit,unit,8}
      {f=8,c=5}       --->   {unit,unit,5,unit,unit,8}
```

Note that each field appears in the position associated with its label, that the size of the tuple is determined by the label with the highest position, and that we fill in unused positions with `unit`.

We do exactly the same thing with record types:

```
      {a:Nat, b:Nat}      --->   {Nat,Nat}
      {c:Nat, a:Nat}      --->   {Nat,Unit,Nat}
      {f:Nat,c:Nat}       --->   {Unit,Unit,Nat,Unit,Unit,Nat}
```

Finally, record projection is encoded as a tuple projection from the appropriate position:

```
      t.l  --->  t.(position of l)
```

It is not hard to check that all the typing rules for the original "direct" presentation of records are validated by this encoding. (The reduction rules are "almost validated" — not quite, because the encoding reorders fields.)

Of course, this encoding will not be very efficient if we happen to use a record with label `foo`! But things are not actually as bad as they might seem: for example, if we assume that our compiler can see the whole program at the same time, we can *choose* the numbering of labels so that we assign small positions to the most frequently used labels. Indeed, there are industrial compilers that essentially do this!

Variants (Optional)

Just as products can be generalized to records, sums can be generalized to n-ary labeled types called *variants*. Instead of T_1+T_2, we can write something like $<l_1:T_1,l_2:T_2,...ln:Tn>$ where l_1,l_2,... are field labels which are used both to build instances and as case arm labels.

These n-ary variants give us almost enough mechanism to build arbitrary inductive data types like lists and trees from scratch — the only thing missing is a way to allow *recursion* in type definitions. We won't cover this here, but detailed treatments can be found in many textbooks — e.g., Types and Programming Languages [Pierce 2002].

Exercise: Formalizing the Extensions

Exercise: 5 stars (STLC_extensions)

In this exercise, you will formalize some of the extensions described in this chapter. We've provided the necessary additions to the syntax of terms and types, and we've included a few examples that you can test your definitions with to make sure they are working as expected. You'll fill in the rest of the definitions and extend all the proofs accordingly.

To get you started, we've provided implementations for:

- numbers
- sums
- lists
- unit

You need to complete the implementations for:

- pairs
- let (which involves binding)
- `fix`

A good strategy is to work on the extensions one at a time, in two passes, rather than trying to work through the file from start to finish in a single pass. For each definition or proof, begin by reading carefully through the parts that are provided for you, referring to the text in the Stlc chapter for high-level intuitions and the embedded comments for detailed mechanics.

```
Module STLCExtended.
```

Syntax

```
Inductive ty : Type :=
  | TArrow : ty → ty → ty
  | TNat : ty
  | TUnit : ty
  | TProd : ty → ty → ty
  | TSum : ty → ty → ty
  | TList : ty → ty.

Inductive tm : Type :=
  (* pure STLC *)
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  (* numbers *)
  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm
  (* pairs *)
  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm
  (* units *)
  | tunit : tm
  (* let *)
  | tlet : string → tm → tm → tm
          (* i.e., let x = t₁ in t₂ *)
  (* sums *)
  | tinl : ty → tm → tm
  | tinr : ty → tm → tm
  | tcase : tm → string → tm → string → tm → tm
          (* i.e., case t₀ of inl x₁ ⇒ t₁ | inr x₂ ⇒ t₂ *)
  (* lists *)
  | tnil : ty → tm
  | tcons : tm → tm → tm
  | tlcase : tm → tm → string → string → tm → tm
           (* i.e., lcase t₁ of | nil ⇒ t₂ | x::y ⇒ t₃ *)
  (* fix *)
  | tfix : tm → tm.
```

Note that, for brevity, we've omitted booleans and instead provided a single $\texttt{if}_0$ form combining a zero test and a conditional. That is, instead of writing

```
       if x = 0 then ... else ...
```

we'll write this:

```
       if₀ x then ... else ...
```

Substitution

```
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)
  | tnat n ⇒
      tnat n
  | tsucc t₁ ⇒
      tsucc (subst x s t₁)
  | tpred t₁ ⇒
      tpred (subst x s t₁)
  | tmult t₁ t₂ ⇒
      tmult (subst x s t₁) (subst x s t₂)
  | tif0 t₁ t₂ t₃ ⇒
      tif0 (subst x s t₁) (subst x s t₂) (subst x s t₃)
  (* FILL IN HERE *)
  | tunit ⇒ tunit
  (* FILL IN HERE *)
  | tinl T t₁ ⇒
      tinl T (subst x s t₁)
  | tinr T t₁ ⇒
```

```
        tinr T (subst x s t1)
    | tcase t0 y1 t1 y2 t2 ⇒
        tcase (subst x s t0)
           y1 (if beq_string x y1 then t1 else (subst x s t1))
           y2 (if beq_string x y2 then t2 else (subst x s t2))
    | tnil T ⇒
        tnil T
    | tcons t1 t2 ⇒
        tcons (subst x s t1) (subst x s t2)
    | tlcase t1 t2 y1 y2 t3 ⇒
        tlcase (subst x s t1) (subst x s t2) y1 y2
          (if beq_string x y1 then
             t3
           else if beq_string x y2 then t3
                else (subst x s t3))
    (* FILL IN HERE *)
    | _ ⇒ t (* ... and delete this line *)
    end.

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

Reduction

Next we define the values of our language.

```
  Inductive value : tm → Prop :=
    | v_abs : ∀ x T11 t12,
        value (tabs x T11 t12)
    (* Numbers are values: *)
    | v_nat : ∀ n1,
        value (tnat n1)
    (* A pair is a value if both components are: *)
    | v_pair : ∀ v1 v2,
        value v1 →
        value v2 →
        value (tpair v1 v2)
    (* A unit is always a value *)
    | v_unit : value tunit
    (* A tagged value is a value:  *)
    | v_inl : ∀ v T,
        value v →
        value (tinl T v)
    | v_inr : ∀ v T,
        value v →
        value (tinr T v)
    (* A list is a value iff its head and tail are values: *)
    | v_lnil : ∀ T, value (tnil T)
    | v_lcons : ∀ v1 vl,
        value v1 →
        value vl →
        value (tcons v1 vl).
```

```
Hint Constructors value.

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

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tapp (tabs x T11 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')
  (* nats *)
  | ST_Succ1 : ∀ t1 t1',
       t1 ==> t1' →
       (tsucc t1) ==> (tsucc t1')
  | ST_SuccNat : ∀ n1,
       (tsucc (tnat n1)) ==> (tnat (S n1))
  | ST_Pred : ∀ t1 t1',
       t1 ==> t1' →
       (tpred t1) ==> (tpred t1')
  | ST_PredNat : ∀ n1,
       (tpred (tnat n1)) ==> (tnat (pred n1))
  | ST_Mult1 : ∀ t1 t1' t2,
       t1 ==> t1' →
       (tmult t1 t2) ==> (tmult t1' t2)
  | ST_Mult2 : ∀ v1 t2 t2',
       value v1 →
       t2 ==> t2' →
       (tmult v1 t2) ==> (tmult v1 t2')
  | ST_MultNats : ∀ n1 n2,
       (tmult (tnat n1) (tnat n2)) ==> (tnat (mult n1 n2))
  | ST_If01 : ∀ t1 t1' t2 t3,
       t1 ==> t1' →
       (tif0 t1 t2 t3) ==> (tif0 t1' t2 t3)
  | ST_If0Zero : ∀ t2 t3,
       (tif0 (tnat 0) t2 t3) ==> t2
  | ST_If0Nonzero : ∀ n t2 t3,
       (tif0 (tnat (S n)) t2 t3) ==> t3
  (* pairs *)
  (* FILL IN HERE *)
  (* let *)
  (* FILL IN HERE *)
  (* sums *)
  | ST_Inl : ∀ t1 t1' T,
       t1 ==> t1' →
        (tinl T t1) ==> (tinl T t1')
  | ST_Inr : ∀ t1 t1' T,
        t1 ==> t1' →
        (tinr T t1) ==> (tinr T t1')
  | ST_Case : ∀ t0 t0' x1 t1 x2 t2,
        t0 ==> t0' →
        (tcase t0 x1 t1 x2 t2) ==> (tcase t0' x1 t1 x2 t2)
  | ST_CaseInl : ∀ v0 x1 t1 x2 t2 T,
        value v0 →
        (tcase (tinl T v0) x1 t1 x2 t2) ==> [x1:=v0]t1
  | ST_CaseInr : ∀ v0 x1 t1 x2 t2 T,
        value v0 →
        (tcase (tinr T v0) x1 t1 x2 t2) ==> [x2:=v0]t2
  (* lists *)
  | ST_Cons1 : ∀ t1 t1' t2,
       t1 ==> t1' →
       (tcons t1 t2) ==> (tcons t1' t2)
  | ST_Cons2 : ∀ v1 t2 t2',
       value v1 →
       t2 ==> t2' →
       (tcons v1 t2) ==> (tcons v1 t2')
  | ST_Lcase1 : ∀ t1 t1' t2 x1 x2 t3,
       t1 ==> t1' →
       (tlcase t1 t2 x1 x2 t3) ==> (tlcase t1' t2 x1 x2 t3)
  | ST_LcaseNil : ∀ T t2 x1 x2 t3,
       (tlcase (tnil T) t2 x1 x2 t3) ==> t2
  | ST_LcaseCons : ∀ v1 vl t2 x1 x2 t3,
       value v1 →
       value vl →
       (tlcase (tcons v1 vl) t2 x1 x2 t3) ==> (subst x2 vl (subst
x1 v1 t3))
  (* fix *)
  (* FILL IN HERE *)

where "t1 '==>' t2" := (step t1 t2).

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

Hint Constructors step.
```

Typing

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

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above.

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

Inductive has_type : context → tm → ty → Prop :=
  (* Typing rules for proper terms *)
  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      Gamma |- (tvar x) ∈ T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      (update 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
  (* nats *)
  | T_Nat : ∀ Gamma n1,
      Gamma |- (tnat n1) ∈ TNat
  | T_Succ : ∀ Gamma t1,
      Gamma |- t1 ∈ TNat →
      Gamma |- (tsucc t1) ∈ TNat
  | T_Pred : ∀ Gamma t1,
      Gamma |- t1 ∈ TNat →
      Gamma |- (tpred t1) ∈ TNat
  | T_Mult : ∀ Gamma t1 t2,
      Gamma |- t1 ∈ TNat →
      Gamma |- t2 ∈ TNat →
      Gamma |- (tmult t1 t2) ∈ TNat
  | T_If0 : ∀ Gamma t1 t2 t3 T1,
      Gamma |- t1 ∈ TNat →
      Gamma |- t2 ∈ T1 →
      Gamma |- t3 ∈ T1 →
      Gamma |- (tif0 t1 t2 t3) ∈ T1
  (* pairs *)
  (* FILL IN HERE *)
  (* unit *)
  | T_Unit : ∀ Gamma,
      Gamma |- tunit ∈ TUnit
  (* let *)
  (* FILL IN HERE *)
  (* sums *)
  | T_Inl : ∀ Gamma t1 T1 T2,
      Gamma |- t1 ∈ T1 →
      Gamma |- (tinl T2 t1) ∈ (TSum T1 T2)
  | T_Inr : ∀ Gamma t2 T1 T2,
      Gamma |- t2 ∈ T2 →
      Gamma |- (tinr T1 t2) ∈ (TSum T1 T2)
  | T_Case : ∀ Gamma t0 x1 T1 t1 x2 T2 t2 T,
      Gamma |- t0 ∈ (TSum T1 T2) →
      (update Gamma x1 T1) |- t1 ∈ T →
      (update Gamma x2 T2) |- t2 ∈ T →
      Gamma |- (tcase t0 x1 t1 x2 t2) ∈ T
  (* lists *)
  | T_Nil : ∀ Gamma T,
      Gamma |- (tnil T) ∈ (TList T)
  | T_Cons : ∀ Gamma t1 t2 T1,
      Gamma |- t1 ∈ T1 →
      Gamma |- t2 ∈ (TList T1) →
      Gamma |- (tcons t1 t2) ∈ (TList T1)
  | T_Lcase : ∀ Gamma t1 T1 t2 x1 x2 t3 T2,
      Gamma |- t1 ∈ (TList T1) →
      Gamma |- t2 ∈ T2 →
      (update (update Gamma x2 (TList T1)) x1 T1) |- t3 ∈ T2 →
      Gamma |- (tlcase t1 t2 x1 x2 t3) ∈ T2
  (* fix *)
  (* FILL IN HERE *)

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

Hint Constructors has_type.
```

Examples

This section presents formalized versions of the examples from above (plus several more). The ones at the beginning focus on specific features; you can use these to make sure your definition of a given feature is reasonable before moving on to extending the proofs later in the file with the cases relating to this feature. The later examples require all the features together, so you'll need to come back to these when you've got all the definitions filled in.

```
Module Examples.
```

Preliminaries

First, let's define a few variable names:

```
Open Scope string_scope.
Notation x := "x".
Notation y := "y".
Notation a := "a".
Notation f := "f".
Notation g := "g".
Notation l := "l".
Notation k := "k".
Notation i1 := "i1".
Notation i2 := "i2".
Notation processSum := "processSum".
Notation n := "n".
Notation eq := "eq".
Notation m := "m".
Notation evenodd := "evenodd".
Notation even := "even".
Notation odd := "odd".
Notation eo := "eo".
```

Next, a bit of Coq hackery to automate searching for typing derivations. You don't need to understand this bit in detail — just have a look over it so that you'll know what to look for if you ever find yourself needing to make custom extensions to `auto`.

The following `Hint` declarations say that, whenever `auto` arrives at a goal of the form `(Gamma |- (tapp` e_1 e_1`) ∈ T)`, it should consider `eapply T_App`, leaving an existential variable for the middle type T_1, and similar for `lcase`. That variable will then be filled in during the search for type derivations for e_1 and e_2. We also include a hint to "try harder" when solving equality goals; this is useful to automate uses of `T_Var` (which includes an equality as a precondition).

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

Numbers

```
Module Numtest.

(* if0 (pred (succ (pred (2 * 0))) then 5 else 6 *)
Definition test :=
  tif0
    (tpred
      (tsucc
        (tpred
          (tmult
            (tnat 2)
            (tnat 0)))))
    (tnat 5)
    (tnat 6).
```

Remove the comment braces once you've implemented enough of the definitions that you think this should work.

```
(*
Example typechecks :
  empty |- test ∈ TNat.
Proof.
  unfold test.
  (* This typing derivation is quite deep, so we need
     to increase the max search depth of auto from the
     default 5 to 10. *)
  auto 10.
Qed.
Example numtest_reduces :
  test ==>* tnat 5.
Proof.
  unfold test. normalize.
Qed.
*)

End Numtest.
```

Products

```
Module Prodtest.

(* ((5,6),7).fst.snd *)
Definition test :=
  tsnd
    (tfst
      (tpair
        (tpair
          (tnat 5)
          (tnat 6))
        (tnat 7))).

(*
Example typechecks :
  empty |- test ∈ TNat.
Proof. unfold test. eauto 15. Qed.

Example reduces :
  test ==>* tnat 6.
Proof. unfold test. normalize. Qed.
*)

End Prodtest.
```

let

```
Module LetTest.

(* let x = pred 6 in succ x *)
Definition test :=
  tlet
    x
    (tpred (tnat 6))
    (tsucc (tvar x)).

(*
Example typechecks :
  empty |- test ∈ TNat.
Proof. unfold test. eauto 15. Qed.

Example reduces :
  test ==>* tnat 6.
Proof. unfold test. normalize. Qed.
*)
```

```
End LetTest.
```

Sums

```
Module Sumtest1.

(* case (inl Nat 5) of
     inl x => x
   | inr y => y *)

Definition test :=
  tcase (tinl TNat (tnat 5))
    x (tvar x)
    y (tvar y).

(*
Example typechecks :
  empty |- test ∈ TNat.
Proof. unfold test. eauto 15. Qed.

Example reduces :
  test ==>* (tnat 5).
Proof. unfold test. normalize. Qed.
*)

End Sumtest1.

Module Sumtest2.

(* let processSum =
     \x:Nat+Nat.
        case x of
          inl n => n
          inr n => if0 n then 1 else 0 in
   (processSum (inl Nat 5), processSum (inr Nat 5))    *)

Definition test :=
  tlet
    processSum
    (tabs x (TSum TNat TNat)
      (tcase (tvar x)
         n (tvar n)
         n (tif0 (tvar n) (tnat 1) (tnat 0))))
    (tpair
      (tapp (tvar processSum) (tinl TNat (tnat 5)))
      (tapp (tvar processSum) (tinr TNat (tnat 5)))).

(*
Example typechecks :
  empty |- test ∈ (TProd TNat TNat).
Proof. unfold test. eauto 15. Qed.

Example reduces :
  test ==>* (tpair (tnat 5) (tnat 0)).
Proof. unfold test. normalize. Qed.
*)

End Sumtest2.
```

Lists

```
Module ListTest.

(* let l = cons 5 (cons 6 (nil Nat)) in
   lcase l of
     nil => 0
   | x::y => x*x *)

Definition test :=
  tlet l
    (tcons (tnat 5) (tcons (tnat 6) (tnil TNat)))
    (tlcase (tvar l)
       (tnat 0)
       x y (tmult (tvar x) (tvar x))).

(*
Example typechecks :
  empty |- test ∈ TNat.
Proof. unfold test. eauto 20. Qed.

Example reduces :
  test ==>* (tnat 25).
Proof. unfold test. normalize. Qed.
*)

End ListTest.
```

fix

```
Module FixTest1.

(* fact := fix
             (\f:nat->nat.
                \a:nat.
                   if a=0 then 1 else a * (f (pred a))) *)
Definition fact :=
  tfix
    (tabs f (TArrow TNat TNat)
      (tabs a TNat
        (tif0
           (tvar a)
           (tnat 1)
           (tmult
              (tvar a)
              (tapp (tvar f) (tpred (tvar a))))))).
```

(Warning: you may be able to typecheck `fact` but still have some rules wrong!)

```
(*
Example fact_typechecks :
  empty |- fact ∈ (TArrow TNat TNat).
Proof. unfold fact. auto 10.
Qed.
*)
```

```
(*
Example fact_example:
  (tapp fact (tnat 4)) ==>* (tnat 24).
Proof. unfold fact. normalize. Qed.
*)

End FixTest1.

Module FixTest2.

(* map :=
     \g:nat->nat.
       fix
         (\f:nat->nat.
            \l:nat.
               case l of
               |  ->
               | x::l -> (g x)::(f l)) *)
Definition map :=
  tabs g (TArrow TNat TNat)
    (tfix
      (tabs f (TArrow (TList TNat) (TList TNat))
        (tabs l (TList TNat)
          (tlcase (tvar l)
            (tnil TNat)
            a l (tcons (tapp (tvar g) (tvar a))
                          (tapp (tvar f) (tvar l))))))).

(*
(* Make sure you've uncommented the last Hint Extern above... *)
Example map_typechecks :
  empty |- map ∈
    (TArrow (TArrow TNat TNat)
      (TArrow (TList TNat)
        (TList TNat))).
Proof. unfold map. auto 10. Qed.

Example map_example :
  tapp (tapp map (tabs a TNat (tsucc (tvar a))))
         (tcons (tnat 1) (tcons (tnat 2) (tnil TNat)))
  ==>* (tcons (tnat 2) (tcons (tnat 3) (tnil TNat))).
Proof. unfold map. normalize. Qed.
*)

End FixTest2.

Module FixTest3.

(* equal =
      fix
        (\eq:Nat->Nat->Bool.
           \m:Nat. \n:Nat.
             if0 m then (if0 n then 1 else 0)
             else if0 n then 0
             else eq (pred m) (pred n))   *)

Definition equal :=
  tfix
```

```
    (tabs eq (TArrow TNat (TArrow TNat TNat))
      (tabs m TNat
        (tabs n TNat
          (tif0 (tvar m)
            (tif0 (tvar n) (tnat 1) (tnat 0))
            (tif0 (tvar n)
              (tnat 0)
              (tapp (tapp (tvar eq)
                              (tpred (tvar m)))
                      (tpred (tvar n)))))))).

(*
Example equal_typechecks :
  empty |- equal ∈ (TArrow TNat (TArrow TNat TNat)).
Proof. unfold equal. auto 10.
Qed.
*)

(*
Example equal_example1:
  (tapp (tapp equal (tnat 4)) (tnat 4)) ==>* (tnat 1).
Proof. unfold equal. normalize. Qed.
*)

(*
Example equal_example2:
  (tapp (tapp equal (tnat 4)) (tnat 5)) ==>* (tnat 0).
Proof. unfold equal. normalize. Qed.
*)

End FixTest3.

Module FixTest4.

(* let evenodd =
         fix
           (\eo: (Nat->Nat * Nat->Nat).
              let e = \n:Nat. if0 n then 1 else eo.snd (pred n) in
              let o = \n:Nat. if0 n then 0 else eo.fst (pred n) in
              (e,o)) in
    let even = evenodd.fst in
    let odd  = evenodd.snd in
    (even 3, even 4)
*)

Definition eotest :=
  tlet evenodd
    (tfix
      (tabs eo (TProd (TArrow TNat TNat) (TArrow TNat TNat))
        (tpair
          (tabs n TNat
            (tif0 (tvar n)
              (tnat 1)
              (tapp (tsnd (tvar eo)) (tpred (tvar n)))))
          (tabs n TNat
            (tif0 (tvar n)
              (tnat 0)
              (tapp (tfst (tvar eo)) (tpred (tvar n))))))))
    (tlet even (tfst (tvar evenodd))
    (tlet odd (tsnd (tvar evenodd))
    (tpair
      (tapp (tvar even) (tnat 3))
      (tapp (tvar even) (tnat 4)))))).

(*
Example eotest_typechecks :
  empty |- eotest ∈ (TProd TNat TNat).
Proof. unfold eotest. eauto 30.
Qed.
*)

(*
Example eotest_example1:
  eotest ==>* (tpair (tnat 0) (tnat 1)).
Proof. unfold eotest. normalize. Qed.
*)

End FixTest4.

End Examples.
```

Properties of Typing

The proofs of progress and preservation for this enriched system are essentially the same (though of course longer) as for the pure STLC.

Progress

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

Context Invariance

```
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)
  (* nats *)
  | afi_succ : ∀ x t,
     appears_free_in x t →
     appears_free_in x (tsucc t)
  | afi_pred : ∀ x t,
     appears_free_in x t →
     appears_free_in x (tpred t)
  | afi_mult1 : ∀ x t₁ t₂,
     appears_free_in x t₁ →
     appears_free_in x (tmult t₁ t₂)
  | afi_mult2 : ∀ x t₁ t₂,
     appears_free_in x t₂ →
     appears_free_in x (tmult t₁ t₂)
  | afi_if₀₁ : ∀ x t₁ t₂ t₃,
     appears_free_in x t₁ →
     appears_free_in x (tif0 t₁ t₂ t₃)
  | afi_if₀₂ : ∀ x t₁ t₂ t₃,
     appears_free_in x t₂ →
     appears_free_in x (tif0 t₁ t₂ t₃)
  | afi_if₀₃ : ∀ x t₁ t₂ t₃,
     appears_free_in x t₃ →
     appears_free_in x (tif0 t₁ t₂ t₃)
  (* pairs *)
  (* FILL IN HERE *)
  (* let *)
  (* FILL IN HERE *)
  (* sums *)
  | afi_inl : ∀ x t T,
      appears_free_in x t →
      appears_free_in x (tinl T t)
  | afi_inr : ∀ x t T,
      appears_free_in x t →
      appears_free_in x (tinr T t)
  | afi_case0 : ∀ x t₀ x₁ t₁ x₂ t₂,
      appears_free_in x t₀ →
      appears_free_in x (tcase t₀ x₁ t₁ x₂ t₂)
  | afi_case1 : ∀ x t₀ x₁ t₁ x₂ t₂,
      x₁ ≠ x →
      appears_free_in x t₁ →
      appears_free_in x (tcase t₀ x₁ t₁ x₂ t₂)
  | afi_case2 : ∀ x t₀ x₁ t₁ x₂ t₂,
      x₂ ≠ x →
      appears_free_in x t₂ →
      appears_free_in x (tcase t₀ x₁ t₁ x₂ t₂)
  (* lists *)
  | afi_cons1 : ∀ x t₁ t₂,
     appears_free_in x t₁ →
     appears_free_in x (tcons t₁ t₂)
  | afi_cons2 : ∀ x t₁ t₂,
     appears_free_in x t₂ →
     appears_free_in x (tcons t₁ t₂)
  | afi_lcase1 : ∀ x t₁ t₂ y₁ y₂ t₃,
     appears_free_in x t₁ →
```

```
        appears_free_in x (tlcase t1 t2 y1 y2 t3)
  | afi_lcase2 : ∀ x t1 t2 y1 y2 t3,
        appears_free_in x t2 →
        appears_free_in x (tlcase t1 t2 y1 y2 t3)
  | afi_lcase3 : ∀ x t1 t2 y1 y2 t3,
        y1 ≠ x →
        y2 ≠ x →
        appears_free_in x t3 →
        appears_free_in x (tlcase t1 t2 y1 y2 t3)
  (* fix *)
  (* FILL IN HERE *)
.

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

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     (update Gamma x U) |- t ∈ S →
     empty |- v ∈ U →
     Gamma |- ([x:=v]t) ∈ S.
+
```

Preservation

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

End STLCExtended.
```
☐