# Lecture Notes 5
# Simply-typed $\lambda$-calculus

Carlo Angiuli

B522: PL Foundations
February 11, 2026

This lecture introduces a famous programming language known as the *simply-typed $\lambda$-calculus* (STLC) *with finite products*, for which we prove type safety. These lecture notes correspond to Section 8.2 and Chapter 10 of Harper [Har16].

## 1 Syntax

The simply-typed $\lambda$-calculus with finite products builds directly upon our previous two topics: as its name suggests, it has types *and* $\lambda$s. As in the lecture on type safety, we will endow the language with statics (a type system) and dynamics (an operational semantics) and show that these agree (satisfy type safety). Because this language has binding, note that we will consider terms up to $\alpha$-equivalence, our statics will require contexts, and our dynamics will require substitution.

We start by introducing two basic judgments $\tau$ ty and $\Gamma \vdash e$ tm. For the first time, the grammar of types has not only basic types but *type operators* as well.

| | | | | |
|---|---|---|---|---|
| *Types* | $\tau ::=$ | $\texttt{unit}$ | $\texttt{unit}$ | unit type |
| | | $\texttt{arr}(\tau_1, \tau_2)$ | $\tau_1 \to \tau_2$ | function type |
| | | $\texttt{prod}(\tau_1, \tau_2)$ | $\tau_1 \times \tau_2$ | product type |
| *Terms* | $e ::=$ | $\texttt{null}$ | $()$ | nullary tuple |
| | | $\texttt{lambda}(\tau, x.e)$ | $\lambda x : \tau.e$ | $\lambda$-abstraction |
| | | $\texttt{app}(e_1, e_2)$ | $e_1\ e_2$ | application |
| | | $\texttt{pair}(e_1, e_2)$ | $(e_1, e_2)$ | ordered pair |
| | | $\texttt{fst}(e)$ | $\texttt{fst}(e)$ | first projection |
| | | $\texttt{snd}(e)$ | $\texttt{snd}(e)$ | second projection |

As discussed in the previous lecture, by writing the syntax of this language as an ABT, we automatically derive notions of $\alpha$-equivalence, free and bound variables,

and capture-avoiding substitution for terms. Like the untyped $\lambda$-calculus, $\lambda$s are the only source of binding in this language.

## 2 Type system

This time we start with the type system. Just as the $\Gamma \vdash e\ \mathsf{tm}$ judgment requires us to maintain a context of which variables are in scope, our typing judgment will require a context that keeps track not only of which variables are in scope but also what *types* they are assumed to have. These *typing contexts* have the form $x_1 : \tau_1, \ldots, x_n : \tau_n$ (where $\tau_i\ \mathsf{ty}$ for each $i$) and are abbreviated $\Gamma$ as before.

**Definition 5.1** (Type system)**.** For $x_1\ \mathsf{tm}, \ldots, x_n\ \mathsf{tm} \vdash e\ \mathsf{tm}$ and $\tau\ \mathsf{ty}$, we define the judgment $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ ("$e$ has type $\tau$ in context $x_1 : \tau_1, \ldots, x_n : \tau_n$") by the following inference rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{var} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}\ \mathsf{unit}\text{-}\textsc{intro}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e_2 : \tau_1 \to \tau_2}\ {\to}\text{-}\textsc{intro} \qquad \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash f\ e_1 : \tau_2}\ {\to}\text{-}\textsc{elim}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ {\times}\text{-}\textsc{intro} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}(e) : \tau_1}\ {\times}\text{-}\textsc{elim}_1$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}(e) : \tau_2}\ {\times}\text{-}\textsc{elim}_2$$

Because our languages are going to get more and more complex, we will start giving systematic names to typing rules to make it easier to refer to them. In addition to the variable rule (which will be present in all systems), for each type former we have some number of *introduction rules* explaining how to *create* something of such a type, and some number of *elimination rules* explaining how to *use* something of such a type.[9] (These names are optional but often written to the right of the horizontal lines, via `\infer*[right=name]{...}{...}`.)

*Remark* 5.2. The simply-typed $\lambda$-calculus (and many systems we consider this semester) is quite *modular* in the sense that it is possible to selectively add and remove types without disrupting the desirable properties of the system. (Harper

[9] The introduction/elimination distinction is clearest for type operators, because there are only a few primitives that generically apply to all pairs, functions, etc. In contrast, one might imagine including dozens of primitives that create and/or use `nums`.

[Har16] treats function and product types in separate chapters!) However, it is important to add or remove type formers, their introduction forms, their elimination forms, and the relevant operational semantics rules as a package deal.

*Remark* 5.3. The term *simply-typed λ-calculus* (without *finite products*) refers to the subset of this language with function types but not product and/or `unit` types. In general, a language is said to be "simply-typed" if it extends the STLC but omits features such as polymorphism or dependency.

*Exercise* 5.4. That said, it is very important that this language include the `unit` type—or at least some other basic type such as `num` or `bool`—in addition to function and product types. Why?

As before, our type system assigns a unique type to every term.

**Lemma 5.5** (Uniqueness of types)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau = \tau'$.

This time, however, there is a lot to say about why uniqueness of types holds.

- We require the same context $\Gamma$ in both typing judgments. The reason should be clear after a moment's thought: otherwise the variable $x$ could clearly have any type, depending on what type it was declared at.
- It is essential that $\lambda$-abstractions come with a type annotation for their argument. Omitting type annotations would let us derive $\cdot \vdash \lambda x.x : \tau \to \tau$ for any type $\tau$, breaking uniqueness of types.
- It may nevertheless seem that uniqueness of types fails, because in the context $x : \texttt{unit}, x : \texttt{unit} \times \texttt{unit}$ we can derive that $x$ has both the type `unit` and the type $\texttt{unit} \times \texttt{unit}$. Note however that every variable in the context came from a binder, and any use of a variable refers unambiguously to *one* binder, even if we inadvertently gave two binders the same name.

  This apparent failure of uniqueness of types is therefore really just a notational ambiguity, because the binding structure of our term disambiguates which context entry the variable $x$ points to. There are a few ways to address this ambiguity, such as regarding contexts as ordered and restricting the variable rule to point to the *most recent* copy of a given variable.

  We will instead require that **all variables in a given context are distinct**, $\alpha$-varying bound variables if necessary in the $\to$-introduction rule.

Now that the typing judgment has contexts, we can and should also prove that typing satisfies the structural properties of hypothetical judgments. Reflexivity is handled by the VAR rule; exchange is automatic as long as contexts are unordered; we will discuss substitution later; and weakening is stated as follows.

**Lemma 5.6** (Weakening). *If* $\Gamma \vdash e : \tau$ *and* $\tau'$ ty *then* $\Gamma, x : \tau' \vdash e : \tau$.

As discussed above, it is important that $x$ does not already occur in $\Gamma$. Some authors add this as an additional hypothesis to the statement of weakening, but in this class we will regard this as part of what it means to respect $\alpha$-equivalence.

# 3 Operational semantics

Next, we define the operational meaning of programs by again specifying which programs have successfully finished evaluating ($e$ val) and what it means to take a single step of computation ($e \longmapsto e'$). Although our term language now contains variables, we restrict evaluation to *closed* terms (terms in the empty context), because it does not make sense to run a program that contains unbound variables.

**Definition 5.7** (Values). For $\cdot \vdash e$ tm, we define the judgment $e$ val ("$e$ is a value") by the following inference rules.

$$\frac{}{() \text{ val}} \qquad \frac{}{\lambda x : \tau.e \text{ val}} \qquad \frac{v_1 \text{ val} \qquad v_2 \text{ val}}{(v_1, v_2) \text{ val}}$$

Note that each introduction form has a value rule, but details vary: pairs are only values when both subterms are values, but $\lambda$s are always values. (In fact, the subterm of a $\lambda$ has a free variable, so it is not eligible to be a value.)

**Definition 5.8** (Small-step operational semantics). For $\cdot \vdash e$ tm, we define the judgment $e \longmapsto e'$ ("$e$ steps to $e'$") by the following inference rules.

$$\frac{f \longmapsto f'}{f\ e_1 \longmapsto f'\ e_1} \qquad \frac{e_1 \longmapsto e_1'}{(\lambda x : \tau_1.e_2)\ e_1 \longmapsto (\lambda x : \tau_1.e_2)\ e_1'}$$

$$\frac{v_1 \text{ val}}{(\lambda x : \tau_1.e_2)\ v_1 \longmapsto e_2[v_1/x]} \qquad \frac{e_1 \longmapsto e_1'}{(e_1, e_2) \longmapsto (e_1', e_2)} \qquad \frac{v_1 \text{ val} \qquad e_2 \longmapsto e_2'}{(v_1, e_2) \longmapsto (v_1, e_2')}$$

$$\frac{e \longmapsto e'}{\mathtt{fst}(e) \longmapsto \mathtt{fst}(e')} \qquad \frac{e \longmapsto e'}{\mathsf{snd}(e) \longmapsto \mathsf{snd}(e')}$$

$$\frac{v_1 \text{ val} \qquad v_2 \text{ val}}{\mathtt{fst}((v_1, v_2)) \longmapsto v_1} \qquad \frac{v_1 \text{ val} \qquad v_2 \text{ val}}{\mathsf{snd}((v_1, v_2)) \longmapsto v_2}$$

*Remark* 5.9. As before, the rules above can be divided into principal reductions and congruence transitions. In this system, every principal reduction concerns

an elimination form that is applied to an introduction form: application of a $\lambda$ or projection from a pair. Reductions of this form are known as *$\beta$-reductions*.

Once again there are several "sanity check" lemmas to prove here.

**Lemma 5.10** (Finality of values)**.** *If $v$* val *then there is no $e'$ such that $v \longmapsto e'$.*

Using finality of values, we can prove determinacy.

**Lemma 5.11** (Determinacy)**.** *If $e \longmapsto e'$ and $e \longmapsto e''$ then $e' = e''$.*

Although proving determinacy is not difficult, it would have been easy to accidentally define a non-deterministic transition system. In the above system:

- To evaluate a pair, we first evaluate the first component (until it reaches a value), and only then do we evaluate the second component.
- To evaluate `fst` and `snd` we insist on fully evaluating their argument, even though this may involve evaluating the unused component of a pair.
- In an application, we first evaluate the function, then the argument (but only after the function is fully evaluated), then perform the substitution (but only after the function and argument are fully evaluated).

Other deterministic evaluation orders are possible too; for example, we could certainly evaluate pairs and applications right-to-left (although left-to-right is common). What is important is that at most one rule applies to any given term.

*Exercise* 5.12. If we replace the two pair-stepping rules above with the following rules, determinacy fails. Why?

$$\frac{e_1 \longmapsto e_1'}{(e_1, e_2) \longmapsto (e_1', e_2)} \qquad \frac{e_2 \longmapsto e_2'}{(e_1, e_2) \longmapsto (e_1, e_2')}$$

*Remark* 5.13. Pairs of overlapping transition rules are called *critical pairs*.

There are other perfectly good deterministic evaluation orders that differ more substantially from ours. For example, we could say that *all* pairs are values, which would involve deleting the two pair-stepping rules, deleting the premises of the pair value rule, and deleting the premises of the `fst` and `snd` principal reductions. For functions, we could similarly $\beta$-reduce as soon as the function is a $\lambda$ rather than evaluating its argument. Such an evaluation strategy is called *call-by-name*, in contrast to our *eager* or *call-by-value* strategy.

**Definition 5.14** (Call-by-name operational semantics)**.** An **alternative** definition of $v$ val and $e \longmapsto e'$ is:

$$\frac{}{() \text{ val}} \qquad \frac{}{\lambda x : \tau . e \text{ val}} \qquad \frac{}{(e_1, e_2) \text{ val}}$$

$$\frac{f \longmapsto f'}{f\ e_1 \longmapsto f'\ e_1} \qquad \frac{}{(\lambda x : \tau_1 . e_2)\ e_1 \longmapsto e_2[e_1/x]} \qquad \frac{e \longmapsto e'}{\mathtt{fst}(e) \longmapsto \mathtt{fst}(e')}$$

$$\frac{e \longmapsto e'}{\mathtt{snd}(e) \longmapsto \mathtt{snd}(e')} \qquad \frac{}{\mathtt{fst}((e_1, e_2)) \longmapsto e_1} \qquad \frac{}{\mathtt{snd}((e_1, e_2)) \longmapsto e_2}$$

Depending on the rest of the language, call-by-value and call-by-name may give some terms significantly different observable behaviors. (What if we have a checked runtime error?) Regardless, they almost always result in different values and evaluation traces. We proceed with eager dynamics for the rest of this lecture.

*Remark* 5.15. No languages implement call-by-name evaluation by default, because it is quite inefficient in practice: the step $(\lambda x : \tau_1 . e_2)\ e_1 \longmapsto e_2[e_1/x]$ replicates the unevaluated $e_1$ many times within $e_2$, potentially forcing us to repeatedly reevaluate $e_1$.[10] A practical improvement is to *memoize* the evaluation of $e_1$ so that it is evaluated at most once;[11] this strategy is called *lazy* or *call-by-need* evaluation, and was independently invented by Henderson and Morris [HM76] and Friedman and Wise [FW76], the latter pair doing so at Indiana University.

Note that memoizing the evaluation of $e_1$ only agrees with the naïve call-by-name strategy if evaluating $e_1$ is idempotent (i.e., evaluating $e_1$ more than once does not have any observable effect on program behavior). The most popular lazy programming language, Haskell [Hud+92], thus pairs laziness with *purity*, or a lack of observable side effects.

Since program evaluation is defined by repeatedly taking single steps until reaching a value, a new important "sanity check" is that stepping always turns a closed term into another closed term.

**Lemma 5.16.** *If* $\cdot \vdash e$ tm *and* $e \longmapsto e'$ *then* $\cdot \vdash e'$ tm.

*Proof.* Straightforward rule induction on $e \longmapsto e'$, with one interesting case:

$$\frac{v_1 \text{ val}}{(\lambda x : \tau_1 . e_2)\ v_1 \longmapsto e_2[v_1/x]}$$

[10] On the other hand, if $x$ does not occur within $e_2$ at all, then call-by-value evaluates $e_1$ once whereas call-by-name evaluates it zero times.

[11] Hackett and Hutton [HH19] recently proposed a more theoretically elegant formulation of this memoization strategy as *clairvoyant call-by-name*.

Showing that the left-hand side being closed implies that the right-hand side is closed boils down to the following fact: "If $x : \tau_1 \vdash e_2$ tm and $\cdot \vdash v_1$ tm then $\cdot \vdash e_2[v_1/x]$ tm." This follows from the substitution lemma for the $\Gamma \vdash e$ tm judgment, which can be proven by a straightforward rule induction. □

# 4 Type safety

Once again, there are some closed terms whose behavior is left unspecified by our operational semantics: terms that neither step to another term nor are values.

*Exercise* 5.17. Give a few different examples of closed terms that are stuck.

We will prove type safety—well-typed terms don't get stuck—which again follows from progress and preservation. Their statements are almost identical to the last time, except that we need to add a few turnstiles.

**Theorem 5.18** (Type safety)**.**

1. *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' : \tau$.
2. *If* $\cdot \vdash e : \tau$ *then either* $e$ val *or* $e \longmapsto e'$.

*Exercise* 5.19. If we're only interested in well-typed *closed* terms, then why did we define the typing judgment for arbitrary contexts?

As before, we need a canonical forms lemma (Lemma 5.20) in order to prove progress (Lemma 5.22). This time, because our operational semantics uses substitution, we will also need a typed substitution lemma (Lemma 5.23) in order to prove preservation (Lemma 5.24).

**Lemma 5.20** (Canonical forms)**.** *Suppose* $\cdot \vdash v : \tau$ *and* $v$ val. *Then:*

1. *If* $\tau = \mathtt{unit}$, *then* $v = ()$.
2. *If* $\tau = \tau_1 \to \tau_2$, *then* $v = \lambda x : \tau_1.e_2$ *where* $x : \tau_1 \vdash e_2 : \tau_2$.
3. *If* $\tau = \tau_1 \times \tau_2$, *then* $v = (v_1, v_2)$ *where* $v_1$ val, $v_2$ val, $\cdot \vdash v_1 : \tau_1$, *and* $\cdot \vdash v_2 : \tau_2$.

*Proof.* By inversion on $\cdot \vdash v : \tau$ and $v$ val. □

*Exercise* 5.21. What does the canonical forms lemma tell us about values of type $\mathtt{unit} \times \mathtt{unit}$? What about values of type $\mathtt{unit} \to \mathtt{unit}$? What about values of type $\mathtt{unit} \times (\mathtt{unit} \to \mathtt{unit})$?

**Lemma 5.22** (Progress)**.** *If* $\cdot \vdash e : \tau$ *then either* $e$ val *or* $e \longmapsto e'$ *for some* $e'$.

*Proof.* By rule induction on $\cdot \vdash e : \tau$.

- Case $\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$ VAR :

  This case cannot happen in the empty context.

- Cases $\dfrac{}{\cdot \vdash () : \mathtt{unit}}$ unit-INTRO , $\dfrac{x : \tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \lambda x : \tau_1 . e_2 : \tau_1 \to \tau_2}$ →-INTRO :

  These are values.

- Case $\dfrac{\cdot \vdash f : \tau_1 \to \tau_2 \qquad \cdot \vdash e_1 : \tau_1}{\cdot \vdash f\ e_1 : \tau_2}$ →-ELIM :

  We show that $f\ e_1 \longmapsto e'$ for some $e'$. ($f\ e_1$ is never a value.) By our first inductive hypothesis, either $f$ val or $f \longmapsto f'$. In the latter case, $f\ e_1 \longmapsto f'\ e_1$, completing the proof.

  In the former case where $f$ val, by Lemma 5.20 we have $f = \lambda x : \tau_1 . e_2$. By our second inductive hypothesis, either $e_1$ val or $e_1 \longmapsto e_1'$. In the first subcase, $(\lambda x : \tau_1 . e_2)\ e_1 \longmapsto e_2[e_1/x]$. In the second subcase, $(\lambda x : \tau_1 . e_2)\ e_1 \longmapsto (\lambda x : \tau_1 . e_2)\ e_1'$.

- Case $\dfrac{\cdot \vdash e_1 : \tau_1 \qquad \cdot \vdash e_2 : \tau_2}{\cdot \vdash (e_1, e_2) : \tau_1 \times \tau_2}$ ×-INTRO :

  We must show that $(e_1, e_2)$ is a value or takes a step, and we know that each of $e_1$ and $e_2$ is a value or takes a step.

  - If $e_1 \longmapsto e_1'$, then $(e_1, e_2) \longmapsto (e_1', e_2)$.
  - If $e_1$ val and $e_2 \longmapsto e_2'$, then $(e_1, e_2) \longmapsto (e_1, e_2')$.
  - If $e_1$ val and $e_2$ val, then $(e_1, e_2)$ val.

- Case $\dfrac{\cdot \vdash e : \tau_1 \times \tau_2}{\cdot \vdash \mathtt{fst}(e) : \tau_1}$ ×-ELIM$_1$ :

  By our inductive hypothesis, $e$ val or $e \longmapsto e'$. In the former case, by Lemma 5.20 we have $e = (v_1, v_2)$ so $\mathtt{fst}(e) \longmapsto v_1$. In the latter case, $\mathtt{fst}(e) \longmapsto \mathtt{fst}(e')$.

- Case $\dfrac{\cdot \vdash e : \tau_1 \times \tau_2}{\cdot \vdash \mathtt{snd}(e) : \tau_2}$ ×-ELIM$_2$ :

  Similar to previous case. □

**Lemma 5.23** (Substitution). *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x] : \tau$.

*Proof.* By rule induction on $\Gamma, x : \tau' \vdash e : \tau$.

- For the VAR rule there are two separate cases, depending on whether the variable $x$ being substituted is the same as the variable term.

  - Case $\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$ VAR :

    In this case the variable in the VAR rule is $x$, so $e = x$ and $\tau = \tau'$. We have $\Gamma, x : \tau \vdash x : \tau$ and $\Gamma \vdash e' : \tau$ and want to show $\Gamma \vdash x[e'/x] : \tau$. But $x[e'/x] = e'$ so this holds by assumption.

  - Case $\dfrac{}{\Gamma', x : \tau', y : \tau \vdash y : \tau}$ VAR :

    In this case the variable in the VAR rule is $y \neq x$, so $e = y$ and $\Gamma = (\Gamma', y : \tau)$. We have $\Gamma', x : \tau', y : \tau \vdash y : \tau$ and $\Gamma', y : \tau \vdash e' : \tau'$ and want to show $\Gamma', y : \tau \vdash y[e'/x] : \tau$. But $y[e'/x] = y$ (because $y \neq x$) so this holds by the VAR rule.

- Case $\dfrac{}{\Gamma, x : \tau' \vdash () : \mathtt{unit}}$ unit-INTRO :

  We assume $\Gamma \vdash e' : \tau'$ and must show $\Gamma \vdash ()[e'/x] : \mathtt{unit}$. But $()[e'/x] = ()$ so this holds by unit-INTRO.

- Case $\dfrac{\Gamma, x : \tau', y : \tau_1 \vdash e_2 : \tau_2}{\Gamma, x : \tau' \vdash \lambda y : \tau_1 . e_2 : \tau_1 \to \tau_2}$ $\to$-INTRO :

  We assume $\Gamma \vdash e' : \tau'$ and must show $\Gamma \vdash (\lambda y : \tau_1 . e_2)[e'/x] : \tau_1 \to \tau_2$. By the IH we have $\Gamma, y : \tau_1 \vdash e_2[e'/x] : \tau_2$. We can choose a fresh name for the binder $y$ (such that $y \neq x$ and $y \notin \mathrm{fv}(e')$), in which case $(\lambda y : \tau_1 . e_2)[e'/x] = \lambda y : \tau_1 . (e_2[e'/x])$. Thus we must show $\Gamma \vdash \lambda y : \tau_1 . (e_2[e'/x]) : \tau_1 \to \tau_2$, which follows from the IH and $\to$-INTRO.

- Case $\dfrac{\Gamma, x : \tau' \vdash f : \tau_1 \to \tau_2 \qquad \Gamma, x : \tau' \vdash e_1 : \tau_1}{\Gamma, x : \tau' \vdash f\ e_1 : \tau_2}$ $\to$-ELIM :

  Assume $\Gamma \vdash e' : \tau'$ and show $\Gamma \vdash (f\ e_1)[e'/x] : \tau_2$. But $(f\ e_1)[e'/x] = (f[e'/x])\ (e_1[e'/x])$ so this follows from the two IHs and $\to$-ELIM.

- Case $\dfrac{\Gamma, x : \tau' \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau' \vdash e_2 : \tau_2}{\Gamma, x : \tau' \vdash (e_1, e_2) : \tau_1 \times \tau_2}$ $\times$-INTRO :

  Similar to previous case.

- Case $\dfrac{\Gamma, x : \tau' \vdash e : \tau_1 \times \tau_2}{\Gamma, x : \tau' \vdash \mathtt{fst}(e) : \tau_1}$ $\times$-ELIM$_1$ :

  Similar to previous case.

- Case $\dfrac{\Gamma, x : \tau' \vdash e : \tau_1 \times \tau_2}{\Gamma, x : \tau' \vdash \mathsf{snd}(e) : \tau_2}$ $\times$-ELIM$_2$ :

  Similar to previous case. □

**Lemma 5.24** (Preservation). *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' : \tau$.

*Proof.* By rule induction on $e \longmapsto e'$.

- Case $\dfrac{f \longmapsto f'}{f\ e_1 \longmapsto f'\ e_1}$:

  By inversion, the only way $\cdot \vdash f\ e_1 : \tau$ can hold is for $\cdot \vdash f : \tau_1 \to \tau$ and $\cdot \vdash e_1 : \tau_1$ to hold. By the IH, $\cdot \vdash f' : \tau_1 \to \tau$, so by $\to$-ELIM, $\cdot \vdash f'\ e_1 : \tau$.

- Case $\dfrac{e_1 \longmapsto e_1'}{(\lambda x : \tau_1.e_2)\ e_1 \longmapsto (\lambda x : \tau_1.e_2)\ e_1'}$:

  Similar to previous case.

- Case $\dfrac{v_1\ \mathsf{val}}{(\lambda x : \tau_1.e_2)\ v_1 \longmapsto e_2[v_1/x]}$:

  By inversion, the only way $\cdot \vdash (\lambda x : \tau_1.e_2)\ v_1 : \tau$ can hold is for $x : \tau_1 \vdash e_2 : \tau$ and $\cdot \vdash v_1 : \tau_1$ to hold. By Lemma 5.23 (Substitution), $\cdot \vdash e_2[v_1/x] : \tau$.

- Case $\dfrac{e_1 \longmapsto e_1'}{(e_1, e_2) \longmapsto (e_1', e_2)}$:

  By inversion, $\tau = \tau_1 \times \tau_2$ and $\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$. By the IH, $\cdot \vdash e_1' : \tau_1$, so by $\times$-INTRO, $\cdot \vdash (e_1', e_2) : \tau_1 \times \tau_2$.

- Case $\dfrac{v_1\ \mathsf{val} \qquad e_2 \longmapsto e_2'}{(v_1, e_2) \longmapsto (v_1, e_2')}$:

  Similar to previous case.

- Case $\dfrac{e \longmapsto e'}{\mathtt{fst}(e) \longmapsto \mathtt{fst}(e')}$:

  By inversion, $\cdot \vdash e : \tau \times \tau_2$. By IH, $\cdot \vdash e' : \tau \times \tau_2$. By $\times$-ELIM$_1$, $\cdot \vdash \mathtt{fst}(e) : \tau$.

- Case $\dfrac{e \longmapsto e'}{\mathsf{snd}(e) \longmapsto \mathsf{snd}(e')}$:

  Similar to previous case.

- Cases $\dfrac{v_1 \text{ val} \qquad v_2 \text{ val}}{\mathtt{fst}((v_1, v_2)) \longmapsto v_1}$,

  By inversion, $\cdot \vdash v_1 : \tau$ (and $\cdot \vdash v_2 : \tau_2$).

- Case $\dfrac{v_1 \text{ val} \qquad v_2 \text{ val}}{\mathsf{snd}((v_1, v_2)) \longmapsto v_2}$:

  Similar to previous case. □

This completes our proof of type safety. It is additionally true that every well-typed program in the STLC terminates, but we will not be able to prove this using techniques we have learned so far in class. We will return to this theorem later.

**Theorem 5.25** (Termination)**.** *If* $\cdot \vdash e : \tau$ *then* $e \longmapsto^* v$ *where* $v$ val.

*Exercise* 5.26. The hypothesis $\cdot \vdash e : \tau$ is necessary. Give an example of an ill-typed term that does not terminate.

## References


[FW76] Daniel P. Friedman and David S. Wise. "CONS Should Not Evaluate its Arguments". In: *Third International Colloquium on Automata, Languages and Programming (ICALP), University of Edinburgh, UK, July 20-23, 1976*. Ed. by S. Michaelson and Robin Milner. Edinburgh University Press, 1976, pp. 257–284. URL: https://hdl.handle.net/2022/33858.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[HH19] Jennifer Hackett and Graham Hutton. "Call-by-need is clairvoyant call-by-value". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (July 2019). DOI: 10.1145/3341718.

[HM76] Peter Henderson and James H. Morris. "A lazy evaluator". In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. POPL '76. Atlanta, Georgia: Association for Computing Machinery, 1976, pp. 95–103. ISBN: 9781450374774. DOI: 10.1145/800168.811543.

[Hud+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. "Report on the Programming Language Haskell: a Non-strict, Purely Functional Language, Version 1.2". In: *SIGPLAN Notices* 27.5 (May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699.