Chapter 4

The FUN Language [Lecture 7–8]

Version of February 26, 2024

Summary We present the call-by-value (CBV, also known as *eager*) big-step and small-step semantics of (a slightly modified variant of) the higher-order functional language FUN from *FSoPL*, and sketch the proof of their equivalence. We then introduce a simple type system for FUN and prove its soundness by means of Progress and Preservation lemmas. Finally, we show that evaluation of well-typed terms without explicit recursion always terminates.

4.1 Big-step and small-step semantics of FUN

4.1.1 Syntax

The syntax of FUN *terms* is given by the following grammar:

$$\begin{aligned} t \;::=\; & \overline{n} \mid \mathbf{true} \mid \mathbf{false} \mid x \mid t_0 + t_1 \mid t_0 \leq t_1 \mid \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\ & \mid (t_1, t_2) \mid \mathbf{fst}(t_0) \mid \mathbf{snd}(t_0) \mid \lambda x.\, t_0 \mid t_1\, t_2 \mid \mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \mid \mathbf{rec}\ x.\, t_0 \end{aligned}$$

Here, like in IMP, n ranges over the integers, while x (and sometimes also y and z, to avoid too many decorations) range over an infinite set of *variables*, analogous to the locations of IMP, but immutable. (In concrete programs, where confusion is unlikely, we may also use x, t, n, etc. as the names of *specific* variables.) Note that, for now, we consider a syntax without any typing constraints. Also, unlike in *FSoPL*'s variant of FUN, we have introduced separate constants for truth values, instead of overloading the integers for this purpose (with zero representing truth, and non-zero falsehood!) Finally, we have added a single integer-comparison operator, from which we can easily construct an equality test on numbers as syntactic sugar:

$$(t_0 = t_1) \stackrel{\text{def}}{\equiv} \mathbf{if}\ t_0 \leq t_1\ \mathbf{then}\ t_1 \leq t_0\ \mathbf{else}\ \mathbf{false}\,.$$

(This should also suggest how we can use **if**-expressions to define $t_0 \wedge t_1$, $t_0 \vee t_1$, and $\neg t_0$.) In the absence of parentheses, in concrete terms, function application groups tighter than addition, which again groups tighter than comparison; both application and addition associate to the left, while comparison is non-associative. Finally, a λ-, **let**, or **rec**-binding

extends as far to the right as syntactically possible; for example, the term $\lambda x.\, f\, x\, y + z$ parses like $\lambda x.\, (((f\, x)\, y) + z)$.

We write $FV(t)$ for the set of *free variables* of t: those that have at least one *free occurrence* in t, i.e., not governed by some inner binding for that variable. This can be formally defined by induction on the syntax of t: for example, $FV(x) = \{x\}$, $FV(t_1\, t_2) = FV(t_1) \cup FV(t_2)$, and $FV(\lambda x.t_0) = FV(t_0) \setminus \{x\}$.

The syntax of *canonical forms* (or *values*) c is as follows:

$$c ::= \overline{n} \mid \textbf{true} \mid \textbf{false} \mid (c_1, c_2) \mid \lambda x.\, t_0\,,$$

where the λ-abstraction body t_0 may only contain x as a free variable. In particular, every c is also a well-formed *closed* term (i.e., with no free variables).

4.1.2 Big-step semantics

The big-step evaluation judgment, $t \downarrow c$, where t must be closed, is defined by the rules in Figure 4.1. In the rules E-App, E-Let, and E-Rec, the notation $t[t'/x]$ denotes substitution of the term t' for all *free* occurrences of x in t. Syntactically, substitution groups even tighter than application. It is defined straightforwardly by induction on the structure of t; in particular, we have:

$$\begin{aligned}
\overline{n}[t'/x] &= \overline{n} \\
y[t'/x] &= \begin{cases} t' & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
(\lambda y.\, t_0)[t'/x] &= \begin{cases} \lambda y.\, t_0 & \text{if } y = x \\ \lambda y.\, t_0[t'/x] & \text{if } y \neq x \text{ (and } y \notin FV(t')) \end{cases} \\
(t_1\, t_2)[t'/x] &= t_1[t'/x]\; t_2[t'/x] \\
&\vdots
\end{aligned}$$

Note that we will only substitute *closed* terms t' for variables, so we do not need to worry about the potential capture of free occurrences of y in t', in the second subcase for λ-abstractions.

The syntax and semantics of FUN are generally what we would expect for an eager functional language in the style of ML, F#, or Scheme. The only slightly unusual aspect is the handling of recursion using the special form **rec** $x.\, t_0$, which behaves just like t_0, but with all free occurrences of x in t_0 replaced by the whole **rec**-term. Almost always, t_0 will itself be a λ-abstraction; for example, assuming for simplicity that the language also includes a multiplication operator (which is itself definable in a similar manner), we could express the usual factorial function as the following term:

$$FAC \equiv \textbf{rec}\ f.\, \lambda n.\, \textbf{if}\ n = \overline{0}\ \textbf{then}\ \overline{1}\ \textbf{else}\ n \times f\,(n + \overline{-1})\,.$$

(Here, f and n are *concrete* variable names.) Then the rules E-App, E-Rec, E-Lam, and E-Num together say that the term $FAC\,\overline{5}$ evaluates to some canonical form if

$$\textbf{if}\ \overline{5} = \overline{0}\ \textbf{then}\ \overline{1}\ \textbf{else}\ \overline{5} \times FAC\,(\overline{5} + \overline{-1})$$

Judgment $\boxed{t \downarrow c}$, t closed:

$$\text{E-Num} : \frac{}{\overline{n} \downarrow \overline{n}} \qquad \text{E-True} : \frac{}{\mathbf{true} \downarrow \mathbf{true}} \qquad \text{E-False} : \frac{}{\mathbf{false} \downarrow \mathbf{false}}$$

$$\text{E-Plus} : \frac{t_0 \downarrow \overline{n_0} \qquad t_1 \downarrow \overline{n_1}}{t_0 + t_1 \downarrow \overline{n_0 + n_1}}$$

$$\text{E-LeqT} : \frac{t_0 \downarrow \overline{n_0} \qquad t_1 \downarrow \overline{n_1}}{t_0 \leq t_1 \downarrow \mathbf{true}} (n_0 \leq n_1) \qquad \text{E-LeqF} : \frac{t_0 \downarrow \overline{n_0} \qquad t_1 \downarrow \overline{n_1}}{t_0 \leq t_1 \downarrow \mathbf{false}} (n_0 > n_1)$$

$$\text{E-IfT} : \frac{t_0 \downarrow \mathbf{true} \qquad t_1 \downarrow c}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \downarrow c} \qquad \text{E-IfF} : \frac{t_0 \downarrow \mathbf{false} \qquad t_2 \downarrow c}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \downarrow c}$$

$$\text{E-Pair} : \frac{t_1 \downarrow c_1 \qquad t_2 \downarrow c_2}{(t_1, t_2) \downarrow (c_1, c_2)} \qquad \text{E-Fst} : \frac{t_0 \downarrow (c_1, c_2)}{\mathbf{fst}(t_0) \downarrow c_1} \qquad \text{E-Snd} : \frac{t_0 \downarrow (c_1, c_2)}{\mathbf{snd}(t_0) \downarrow c_2}$$

$$\text{E-Lam} : \frac{}{\lambda x.\, t_0 \downarrow \lambda x.\, t_0} \qquad \text{E-App} : \frac{t_1 \downarrow \lambda x.\, t_0 \qquad t_2 \downarrow c_2 \qquad t_0[c_2/x] \downarrow c}{t_1\, t_2 \downarrow c}$$

$$\text{E-Let} : \frac{t_1 \downarrow c_1 \qquad t_2[c_1/x] \downarrow c}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \downarrow c} \qquad \text{E-Rec} : \frac{t_0[(\mathbf{rec}\ x.\, t_0)/x] \downarrow c}{\mathbf{rec}\ x.\, t_0 \downarrow c}$$

Figure 4.1: Big-step CBV semantics of FUN

evaluates to that same canonical form; and by a few further rule applications, we get that that canonical form must indeed be $\overline{120}$.

We can easily check that the semantics is deterministic (if $t \downarrow c$ and $t \downarrow c'$, then $c = c'$), but not total (e.g., $\mathbf{rec}\ x.\, x$ does not evaluate to anything, and neither does $\overline{5} + \mathbf{true}$.)

4.1.3 Small-step semantics

The small-step reduction judgment, $t \to t'$, is defined by the rules in Figure 4.2. It is likewise deterministic but not total. The multi-step judgment, $t \to^* t'$, is obtained from the single-step one in the usual way.

4.1.4 Equivalence between big-step and small-step semantics

The proof strategy for showing equivalence of the two semantics is essentially the same as for IMP:

Lemma 4.1 *If $t \to^* t'$ and $t' \to^* t''$, then $t \to^* t''$.*

Proof. By induction on the first derivation. ∎

Theorem 4.2 *If $t \downarrow c$, then $t \to^* c$.*

Proof. By induction on the big-step derivation, using Lemma 4.1 in most inductive steps. ∎

Judgment $\boxed{t \to t'}$, t closed.

(no rules for $t = \overline{n}$, **true**, **false**, or $\lambda x.\, t_0$)

$$\text{S-Plus1} : \frac{t_0 \to t_0'}{t_0 + t_1 \to t_0' + t_1} \qquad \text{S-Plus2} : \frac{t_1 \to t_1'}{c_0 + t_1 \to c_0 + t_1'} \qquad \text{S-Plus} : \frac{}{\overline{n_0} + \overline{n_1} \to \overline{n_0 + n_1}}$$

$$\text{S-Leq1} : \frac{t_0 \to t_0'}{t_0 \leq t_1 \to t_0' \leq t_1} \qquad \text{S-Leq2} : \frac{t_1 \to t_1'}{c_0 \leq t_1 \to c_0 \leq t_1'}$$

$$\text{S-LeqT} : \frac{}{\overline{n_0} \leq \overline{n_1} \to \mathbf{true}} (n_0 \leq n_1) \qquad \text{S-LeqF} : \frac{}{\overline{n_0} \leq \overline{n_1} \to \mathbf{false}} (n_0 > n_1)$$

$$\text{S-If1} : \frac{t_0 \to t_0'}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to \mathbf{if}\ t_0'\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2}$$

$$\text{S-IfT} : \frac{}{\mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2 \to t_1} \qquad \text{S-IfF} : \frac{}{\mathbf{if\ false\ then}\ t_1\ \mathbf{else}\ t_2 \to t_2}$$

$$\text{S-Pair1} : \frac{t_1 \to t_1'}{(t_1, t_2) \to (t_1', t_2)} \qquad \text{S-Pair2} : \frac{t_2 \to t_2'}{(c_1, t_2) \to (c_1, t_2')}$$

$$\text{S-Fst1} : \frac{t_0 \to t_0'}{\mathbf{fst}(t_0) \to \mathbf{fst}(t_0')} \qquad \text{S-Fst} : \frac{}{\mathbf{fst}((c_1, c_2)) \to c_1}$$

$$\text{S-Snd1} : \frac{t_0 \to t_0'}{\mathbf{snd}(t_0) \to \mathbf{snd}(t_0')} \qquad \text{S-Snd} : \frac{}{\mathbf{snd}((c_1, c_2)) \to c_2}$$

$$\text{S-App1} : \frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2} \qquad \text{S-App2} : \frac{t_2 \to t_2'}{c_1\, t_2 \to c_1\, t_2'} \qquad \text{S-App} : \frac{}{(\lambda x.\, t_0)\, c_2 \to t_0[c_2/x]}$$

$$\text{S-Let1} : \frac{t_1 \to t_1'}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \to \mathbf{let}\ x \Leftarrow t_1'\ \mathbf{in}\ t_2} \qquad \text{S-Let} : \frac{}{\mathbf{let}\ x \Leftarrow c_1\ \mathbf{in}\ t_2 \to t_2[c_1/x]}$$

$$\text{S-Rec} : \frac{}{\mathbf{rec}\ x.\, t_0 \to t_0[(\mathbf{rec}\ x.\, t_0)/x]}$$

Judgment $\boxed{t \to^* t'}$, t closed.

$$\text{SS-Zero} : \frac{}{t \to^* t} \qquad \text{SS-More} : \frac{t \to t'' \qquad t'' \to^* t'}{t \to^* t'}$$

Figure 4.2: Small-step CBV semantics of FUN

Lemma 4.3 *For any c, $c \downarrow c$.*

Proof. By induction on the structure of c. All but the case $c = (c_1, c_2)$ are immediate, and that one follows directly by the IH and rule E-Pair. ∎

Lemma 4.4 *If $t \to t'$ and $t' \downarrow c$, then $t \downarrow c$.*

Proof. By induction on the first derivation. ∎

Theorem 4.5 *If $t \to^* c$ then $t \downarrow c$.*

Proof. By induction on the derivation. If $t \to^* c$ in zero steps, we must have $t = c$, and thus $t \downarrow c$ follows from Lemma 4.3. Otherwise, we have subderivations showing $t \to t'$ and $t' \to^* c$. By IH on the latter, we obtain $t' \downarrow c$; and Lemma 4.4 on that and the subderivation for the first step immediately gives us $t \downarrow c$. ∎

Note, though, that the equivalence results in Theorems 4.2 and 4.5 only cover the case where the term evaluates successfully. If the evaluation fails for whatever reason, the small-step semantics can make finer distinctions than the big-step one.

4.2 Types and type soundness

In the IMP language, the syntactic distinction between arithmetic and boolean expressions ensured that meaningless combinations, e.g., $\overline{5} + \mathbf{true}$, could not even be written down as well-formed terms. In FUN, however, the syntax does include terms that we would consider erroneous, such as the above, or $(\lambda f.\, f\, \mathbf{true})\, (\lambda x.\, \overline{5} + x)$. One purpose of a type system is to reject such terms even before the program is run.

In the big-step semantics, there is no formal distinction between terms that fail to evaluate to a canonical form because no rule applies to them (such as $\overline{5} + \mathbf{true}$) and those that actively diverge (such as $\mathbf{rec}\ x.\, x$). One can modify the semantics so that terms of the former kind evaluate to some notion of error result, often written **wrong**. Then one can hope to prove that no well-typed term evaluates to **wrong**. However, this approach involves modifying the operational semantics to introduce and propagate errors, and in particular requires us to *enumerate badness*: we must specifically introduce rules for each particular error condition that might arise, only to then prove that none of them actually will.

A small-step semantics, on the other hand, allows for a simple distinction between erroneous and and diverging terms. Let us introduce the following:

Definition 4.6 *A closed term is said to be* stuck *if it is not a canonical form, but also cannot reduce to another term.*

We can think of such stuck terms as runtime errors. Note that stuckness just talks about the current configuration: a term such as $(\lambda x.\, x + \mathbf{true})\, \overline{5}$ is *not* stuck (though it will reduce to a stuck term). On the other hand, the term $(\lambda x.\, \overline{3})\, (\overline{5} + \mathbf{true})$ *is* stuck (in the CBV semantics), because no reduction rule applies to it. (Recall that S-App requires the function argument in an application to be a canonical form, not merely a term that cannot be further reduced.)

Definition 4.7 *A closed term t is called* safe *if for any t' such that $t \to^* t'$ (in particular, including $t' = t$), t' is not stuck.*

Note that a term that reduces forever is also considered safe by this definition. We now want to introduce a type system that ensures that all well-typed terms are safe, i.e., that they will never get stuck.

4.2.1 A simple type system for FUN

(Note: "simple" actually has a technical meaning here; later in the course we will see a couple of non-simple systems that generalize or extend this system in various ways.)

The sort of well-formed simple types for FUN is given by the following grammar:

$$\tau ::= \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

If we had a language with no variables, we could immediately introduce a typing judgment $t : \tau$, expressing that a (necessarily closed) term had a given type. However, to properly reason about terms with variables – specifically, λ-abstractions, **let**-bindings, and recursion – we need to generalize this judgment to also account for the types of variables occurring in the term.

A *typing context* (or *type environment*) Γ is a finite map from variable names to types:

$$\Gamma = [x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n]$$

where all the x_i are distinct. We thus have $\mathrm{dom}\,\Gamma = \{x_1, ..., x_n\}$. The empty typing context is written $[]$ (so $\mathrm{dom}\,[] = \emptyset$). We then define a *typing judgment*, $\Gamma \vdash t : \tau$, "in context Γ, the term t has type τ", by the rules in Figure 4.3. (In the typing rule T-Var, the side condition implicitly also requires that x is in the domain of the function Γ in the first place.) It is easy to see, by induction on the typing derivation, that if $\Gamma \vdash t : \tau$, then $FV(t) \subseteq \mathrm{dom}\,\Gamma$. In particular, if $[] \vdash t : \tau$ for some τ, then t must be closed.

Note that, in this system, a term may have more than one type, even for a fixed Γ. For example, we have $[] \vdash \lambda x.\, x : \mathbf{int} \to \mathbf{int}$ and $[] \vdash \lambda x.\, x : \mathbf{bool} \to \mathbf{bool}$. However, a variable can only be given one type in a typing context, so a term like

$$\begin{array}{l}\mathbf{let}\ f \Leftarrow \lambda x.\, x\ \mathbf{in} \\ \quad \mathbf{if}\ f\,\mathbf{true}\ \mathbf{then}\ f\,\overline{5}\ \mathbf{else}\ \overline{6}\end{array}$$

is *not* typable. (In an extension of the system with *polymorphic* types, variables like f can be given a generic type that can be instantiated differently at each use.)

Somewhat analogously to evaluation, terms may fail to have a type for two reasons: either because there is no applicable rule, or because the only possible rules would assign the term an infinitely large type. A simple example of the former kind is the term $\lambda x.\, x + \mathbf{true}$. Since there is only one rule for λ-abstractions, we would need, for some τ_1 and τ_2, a derivation of the premise of T-Lam:

$$[x \mapsto \tau_1] \vdash x + \mathbf{true} : \tau_2$$

To derive this typing by the only applicable rule (T-Plus), we must have $[x \mapsto \tau_1] \vdash x : \mathbf{int}$ (which is possible if we pick $\tau_1 = \mathbf{int}$), but also that $[x \mapsto \tau_1] \vdash \mathbf{true} : \mathbf{int}$, which is evidently not derivable by the only typing rule for **true**, namely T-True.

Judgment $\boxed{\Gamma \vdash t : \tau}$:

$$\text{T-Var} : \frac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau) \qquad \text{T-Num} : \frac{}{\Gamma \vdash \overline{n} : \mathbf{int}}$$

$$\text{T-True} : \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \text{T-False} : \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}$$

$$\text{T-Plus} : \frac{\Gamma \vdash t_0 : \mathbf{int} \qquad \Gamma \vdash t_1 : \mathbf{int}}{\Gamma \vdash t_0 + t_1 : \mathbf{int}} \qquad \text{T-Leq} : \frac{\Gamma \vdash t_0 : \mathbf{int} \qquad \Gamma \vdash t_1 : \mathbf{int}}{\Gamma \vdash t_0 \leq t_1 : \mathbf{bool}}$$

$$\text{T-If} : \frac{\Gamma \vdash t_0 : \mathbf{bool} \qquad \Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$$

$$\text{T-Pair} : \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2} \qquad \text{T-Fst} : \frac{\Gamma \vdash t_0 : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}(t_0) : \tau_1} \qquad \text{T-Snd} : \frac{\Gamma \vdash t_0 : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}(t_0) : \tau_2}$$

$$\text{T-Lam} : \frac{\Gamma[x \mapsto \tau_1] \vdash t_0 : \tau_2}{\Gamma \vdash \lambda x.\, t_0 : \tau_1 \to \tau_2} \qquad \text{T-App} : \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

$$\text{T-Let} : \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 : \tau_2} \qquad \text{T-Rec} : \frac{\Gamma[x \mapsto \tau] \vdash t_0 : \tau}{\Gamma \vdash \mathbf{rec}\ x.\, t_0 : \tau}$$

Figure 4.3: Typing rules for FUN

An example of the second kind of untypable term is $\lambda x.\, x\, x$. Again, being a λ-abstraction, if it were typable at all, it would need to have a type $\tau_1 \to \tau_2$ where

$$[x \mapsto \tau_1] \vdash x\, x : \tau_2$$

But now, by inspecting the typing rule T-App, we note that the type τ_1 of the x in function position must itself be a function type, $\tau_1 = \tau_{11} \to \tau_{12}$; and for the application to be well-typed, τ_{11} must be the same as the type of the x in argument position, i.e., $\tau_{11} = \tau_1$. In other words, we must have $\tau_1 = \tau_1 \to \tau_{12}$, but that is impossible if τ_1 is of finite size. (It *is* possible to give a type to $\lambda x.\, x\, x$ in some non-simple type systems, such as *impredicative polymorphic* or *equi-recursive* types, however.)

4.2.2 Type soundness

In this section we shall show that every well-typed closed term is safe, i.e., can either be reduced forever without getting stuck, or will eventually reduce to a canonical form. In the latter case, we will additionally be able to conclude that the canonical form is of the same type as the original term, so that, e.g., any term of type **int** can only reduce to a numeral, not to a truth value.

The proof is expressed as two key results, traditionally called the Progress and Preservation lemmas. The former says that a closed (and hence potentially evaluable), well-typed term is never *immediately* stuck:

Lemma 4.8 (Progress) *If $[] \vdash t : \tau$, then either $t = c$ for some canonical form c, or there exists a t' such that $t \to t'$.*

Proof. By induction on the typing derivation $\mathcal{T}$ of $[] \vdash t : \tau$:

- Case $\mathcal{T} = \text{T-VAR}\dfrac{}{[] \vdash x : \tau}$, where $[](x) = \tau$.

  This case cannot actually arise, since x is obviously not in the domain of the empty typing-context map. So the situation where $t = x$ is vacuously covered.

- Case $\mathcal{T} = \text{T-NUM}\dfrac{}{[] \vdash \overline{n} : \mathbf{int}}$. Immediate, since $\overline{n}$ is a canonical form.

- The cases where $\mathcal{T}$ ends in T-TRUE or T-FALSE are analogous to T-NUM.

- Case $\mathcal{T} = \text{T-PLUS}\dfrac{\begin{array}{cc}\mathcal{T}_0 & \mathcal{T}_1 \\ [] \vdash t_0 : \mathbf{int} & [] \vdash t_1 : \mathbf{int}\end{array}}{[] \vdash t_0 + t_1 : \mathbf{int}}$, so $t = t_0 + t_1$.

  By IH on $\mathcal{T}_0$, either $t_0 = c_0$ (with c_0 canonical), or $t_0 \to t_0'$ for some t_0'. In the later case, we get $t_0 + t_1 \to t_0' + t_1$ by S-PLUS1, and we are done, with $t' = t_0' + t_1$; so it remains to consider the case where $t_0 = c_0$. Analogously, by IH on $\mathcal{T}_1$, either t_1 is canonical or $t_1 \to t_1'$ for some t_1'. In the latter case we use S-PLUS2 to show that $c_0 + t_1 \to c_0 + t_1'$, so it remains only to consider the case $t = c_0 + c_1$. But since the only canonical forms of type **int** (as required by $\mathcal{T}_0$ and $\mathcal{T}_1$) are evidently the numerals, we must have $c_0 = \overline{n_0}$ for some integer n_0, and likewise $c_1 = \overline{n_1}$, so $t = \overline{n_0} + \overline{n_1}$. And then $t \to \overline{n_0 + n_1} = t'$ by S-PLUS.

- The case for $\mathcal{T}$ ending in T-LEQ is a analogous to T-PLUS, except that for the final subcase, in which $t = (\overline{n_0} \leq \overline{n_1})$, we exploit that at least one (in fact, exactly one) of the rules S-LEQT or S-LEQF can be used to perform the reduction.

- Case $\mathcal{T} = \text{T-IF}\dfrac{\begin{array}{ccc}\mathcal{T}_0 & \mathcal{T}_1 & \mathcal{T}_2 \\ [] \vdash t_0 : \mathbf{bool} & [] \vdash t_1 : \tau & [] \vdash t_2 : \tau\end{array}}{[] \vdash \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$.

  By IH on $\mathcal{T}_0$, either t_0 is canonical or $t_0 \to t_0'$. In the latter case, the entire **if**-term can take a step by S-IF1, so it remains to consider the case where $t_0 = c_0$. But the only canonical forms of type **bool** (per $\mathcal{T}_0$) are **true** and **false**. In the former subcase, we have $t = \mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2$, which reduces to $t' = t_1$ by rule S-IFT; the subcase $c_0 = \mathbf{false}$ is analogous.

- The case where $\mathcal{T}$ ends in T-PAIR is analogous to T-PLUS or T-LEQ, except that in the final subcase, we use that $t = (c_1, c_2)$ is already a canonical form, rather than showing that it can can take at least one more step.

- Case $\mathcal{T} = \text{T-FST}\dfrac{\begin{array}{c}\mathcal{T}_0 \\ [] \vdash t_0 : \tau_1 \times \tau_2\end{array}}{[] \vdash \mathbf{fst}(t_0) : \tau_1}$, so $t = \mathbf{fst}(t_0)$ and $\tau = \tau_1$.

  By IH on $\mathcal{T}_0$, either t_0 is canonical or $t_0 \to t_0'$. In the latter case, we get $t \to t'$ for $t' = \mathbf{fst}(t_0')$, using S-FST1. In the former, the only canonical form of type $\tau_1 \times \tau_2$ is of shape $t_0 = (c_1, c_2)$, so $t = \mathbf{fst}(t_0)$ reduces to $t' = c_1$ by S-FST. The case for $\mathcal{T}$ ending with T-SND is analogous.

- Case $\mathcal{T} = \text{T-Lam}\dfrac{\begin{array}{c}\mathcal{T}_0\\ [x \mapsto \tau_1] \vdash t_0 : \tau_2\end{array}}{[] \vdash \lambda x.\, t_0 : \tau_1 \to \tau_2}$, so $\tau = \tau_1 \to \tau_2$.

  This case is immediate, since $t = \lambda x.\, t_0$ is already a canonical form. Note that we did not need to use the IH on $\mathcal{T}_0$ here (which we couldn't anyway, since t_0 is not closed).

- Case $\mathcal{T} = \text{T-App}\dfrac{\begin{array}{cc}\mathcal{T}_1 & \mathcal{T}_2\\ [] \vdash t_1 : \tau_1 \to \tau & [] \vdash t_2 : \tau_1\end{array}}{[] \vdash t_1\, t_2 : \tau}$, so $t = t_1\, t_2$.

  By IH on $\mathcal{T}_1$, we have $t_1 = c_1$ or $t_1 \to t_1'$. In the latter case, $t_1\, t_2 \to t_1'\, t_2$ by S-App1, and we are done. In the former, by IH on $\mathcal{T}_2$ we get that either $t_2 = c_2$ or $t_2 \to t_2'$. Again, in the latter case, $t = c_1\, t_2$ can reduce to $t' = c_1\, t_2'$ (by S-App2); and in the former, we must actually have $t_1 = c_1 = \lambda x.\, t_0$ for some x and t_0 (since no other canonical forms have type $\tau_1 \to \tau_2$), so $t = (\lambda x.\, t_0)\, c_2$, which steps to $t' = t_0[c_2/x]$ by rule S-App.

- (The case where $\mathcal{T}$ ends in T-Let is left as an exercise.)

- Case $\mathcal{T} = \text{T-Rec}\dfrac{\begin{array}{c}\mathcal{T}_0\\ [x \mapsto \tau] \vdash t_0 : \tau\end{array}}{[] \vdash \mathbf{rec}\ x.\, t_0 : \tau}$.

  This case is also immediate, because $t = \mathbf{rec}\ x.\, t_0$ can always be reduced to $t' = t_0[(\mathbf{rec}\ x.\, t_0)/x]$ by S-Rec. (Again we didn't need to use $\mathcal{T}_0$ for anything here.)

■

The second part of the type-soundness argument says that a well-typed term always reduces to another well-typed term. For showing that, we'll need an auxiliary lemma, stating that replacing a variable with a closed term of the same type preserves typability. We start by an even simpler property, saying that typability is preserved when we add additional assumptions about variables to the typing context:

Lemma 4.9 (Weakening) *If $\Gamma \vdash t : \tau$ and $\Gamma \subseteq \Gamma'$ (i.e., for all x such that $\Gamma(x)$ is defined, so is $\Gamma'(x)$, and with $\Gamma'(x) = \Gamma(x)$), then also $\Gamma' \vdash t : \tau$.*

Proof. Simple induction on the structure of t. The only interesting case is when t is a variable, in which case the result follows directly from the definition of $\Gamma \subseteq \Gamma'$. For the inductive cases involving variable binding, we also exploit that, as is easily checked, when $\Gamma \subseteq \Gamma'$, then also $\Gamma[x \mapsto \tau] \subseteq \Gamma'[x \mapsto \tau]$, for all x and τ. ■

Lemma 4.10 (Substitution) *If $\Gamma[x \mapsto \tau'] \vdash t : \tau$ and $[] \vdash t' : \tau'$, then $\Gamma \vdash t[t'/x] : \tau$.*

Proof. By induction on the typing derivation $\mathcal{T}$ for t. We show a few representative cases:

- Case $\mathcal{T} = \text{T-Num}\dfrac{}{\Gamma[x \mapsto \tau'] \vdash \overline{n} : \mathbf{int}}$, so $t = \overline{n}$ and $\tau = \mathbf{int}$.

  Since $\overline{n}[t'/x] = \overline{n}$, we get the required derivation $\Gamma \vdash \overline{n} : \mathbf{int}$ by T-Num as well.

- Case $\mathcal{T} = \text{T-Var}\dfrac{}{\Gamma[x \mapsto \tau'] \vdash y : \tau}$, so $t = y$ and $\tau = \Gamma[x \mapsto \tau'](y)$. There are two possibilities here:
  - Subcase $y = x$, so $\tau = \Gamma[x \mapsto \tau'](x) = \tau'$.
    Then $y[t'/x] = x[t'/x] = t'$, so we must show that $\Gamma \vdash t' : \tau$. But that follows directly from the typing assumption on t' and Lemma 4.9 (because $[] \subseteq \Gamma$).
  - Subcase $y \neq x$, so $\tau = \Gamma(y)$.
    Then $y[t'/x] = y$, and $\Gamma \vdash y : \tau$ by T-Var again.

- Case $\mathcal{T} = \text{T-Lam}\dfrac{\overset{\mathcal{T}_0}{(\Gamma[x \mapsto \tau'])[y \mapsto \tau_1] \vdash t_0 : \tau_2}}{\Gamma[x \mapsto \tau'] \vdash \lambda y.\, t_0 : \tau_1 \to \tau_2}$, so $t = \lambda y.\, t_0$ and $\tau = \tau_1 \to \tau_2$, for some τ_1 and τ_2. Again, there are two possibilities:
  - Subcase $y = x$. Then $(\Gamma[x \mapsto \tau'])[y \mapsto \tau_1] = (\Gamma[y \mapsto \tau'])[y \mapsto \tau_1] = \Gamma[y \mapsto \tau_1]$, so $\mathcal{T}_0$ is also already a derivation of $\Gamma[y \mapsto \tau_1] \vdash t_0 : \tau_2$; and thus, by T-Lam, $\Gamma \vdash \lambda y.\, t_0 : \tau_1 \to \tau_2$. But since $(\lambda y.\, t_0)[t'/x] = (\lambda y.\, t_0)[t'/y] = \lambda y.\, t_0$, this means we have also shown $\Gamma \vdash (\lambda y.\, t_0)[t'/x] : \tau$, as required.
  - Subcase $y \neq x$. Here, since $(\Gamma[x \mapsto \tau'])[y \mapsto \tau_1] = (\Gamma[y \mapsto \tau_1])[x \mapsto \tau']$, $\mathcal{T}_0$ is also a derivation of $(\Gamma[y \mapsto \tau_1])[x \mapsto \tau'] \vdash t_0 : \tau_2$, and thus by IH on $\mathcal{T}_0$, $\Gamma[y \mapsto \tau_1] \vdash t_0[t'/x] : \tau_2$, from which T-Lam gives us $\Gamma \vdash \lambda y.\, t_0[t'/x] : \tau_1 \to \tau_2$. But since $(\lambda y.\, t_0)[t'/x] = \lambda y.\, t_0[t'/x]$, this means we have again shown $\Gamma \vdash (\lambda y.\, t_0)[t'/x] : \tau$.

- Case $\mathcal{T} = \text{T-App}\dfrac{\overset{\mathcal{T}_1}{\Gamma[x \mapsto \tau'] \vdash t_1 : \tau_1 \to \tau_2} \quad \overset{\mathcal{T}_2}{\Gamma[x \mapsto \tau'] \vdash t_2 : \tau_1}}{\Gamma[x \mapsto \tau'] \vdash t_1\, t_2 : \tau_2}$, so $t = t_1\, t_2$ and $\tau = \tau_2$.

  By IH on $\mathcal{T}_1$, we get a derivation of $\Gamma \vdash t_1[t'/x] : \tau_1 \to \tau$; and by IH on $\mathcal{T}_2$, a derivation of $\Gamma \vdash t_2[t'/x] : \tau_1$. Putting these together with T-App, we get $\Gamma \vdash t_1[t'/x]\, t_2[t'/x] : \tau$; and since $(t_1\, t_2)[t'/x] = t_1[t'/x]\, t_2[t'/x]$, we are done. ∎

Ultimately, we shall only need the substitution lemma for $\Gamma = []$, but as the proof case for $t = \lambda y.\, t_0$ with $y \neq x$ shows, we need the stronger formulation to make the induction on $\mathcal{T}$ go through, because not all subterms of t are themselves closed.

Note that we could also show an even more general version of the lemma, where we allow $\Gamma \vdash t' : \tau'$, i.e, t' need not be closed. The proof is somewhat more involved, however, because we then need to reason precisely about capture-avoiding substitutions.

Lemma 4.11 (Preservation) *If $t \to t'$ (by $\mathcal{S}$) and $[] \vdash t : \tau$ (by $\mathcal{T}$), then also $[] \vdash t' : \tau$ (by some $\mathcal{T}'$).*

Proof. By induction on the derivation $\mathcal{S}$. (It would also have been possible to do the induction on $\mathcal{T}$, but our approach goes a little smoother.)

- Case $\mathcal{S} = \text{S-PLUS1}\dfrac{\overset{\mathcal{S}_0}{t_0 \to t_0'}}{t_0 + t_1 \to t_0' + t_1}$. Since $t = t_0 + t_1$, the typing derivation $\mathcal{T}$ must be:

$$\mathcal{T} = \text{T-PLUS}\frac{\overset{\mathcal{T}_0}{[] \vdash t_0 : \mathbf{int}} \qquad \overset{\mathcal{T}_1}{[] \vdash t_1 : \mathbf{int}}}{[] \vdash t_0 + t_1 : \mathbf{int}}\,.$$

  Now, by IH on $\mathcal{S}_0$ with $\mathcal{T}_0$, we get a derivation $\mathcal{T}_0'$ of $[] \vdash t_0' : \mathbf{int}$, and thus we can reconstruct the required typing derivation for $t' = t_0' + t_1$:

$$\mathcal{T}' = \text{T-PLUS}\frac{\overset{\mathcal{T}_0'}{[] \vdash t_0' : \mathbf{int}} \qquad \overset{\mathcal{T}_1}{[] \vdash t_1 : \mathbf{int}}}{[] \vdash t_0' + t_1 : \mathbf{int}}\,.$$

- The cases for S-PLUS2, S-LEQ1, S-LEQ2, S-IF1, S-PAIR1, S-PAIR2, S-FST1, S-SND1, S-APP1, and S-APP2 are essentially identical to S-PLUS1: the reduction consists of replacing an immediate subterm of the original term with one of the same type (by the IH), so we can immediately rebuild the new typing derivation $\mathcal{T}'$ from $\mathcal{T}$. It therefore remains to consider just the computation rules.

- Case $\mathcal{S} = \text{S-PLUS}\dfrac{}{\overline{n_0} + \overline{n_1} \to \overline{n_0 + n_1}}$, so $t = \overline{n_0} + \overline{n_1}$ and $t' = \overline{n_0 + n_1}$.

  Here, the typing derivation for t must look like:

$$\mathcal{T} = \text{T-PLUS}\frac{\text{T-NUM}\dfrac{}{[] \vdash \overline{n_0} : \mathbf{int}} \qquad \text{T-NUM}\dfrac{}{[] \vdash \overline{n_1} : \mathbf{int}}}{[] \vdash \overline{n_0} + \overline{n_1} : \mathbf{int}}\,,$$

  so in particular $\tau = \mathbf{int}$. We can then construct the required derivation $\mathcal{T}'$ for t':

$$\mathcal{T}' = \text{T-NUM}\frac{}{[] \vdash \overline{n_0 + n_1} : \mathbf{int}}\,.$$

- The cases where $\mathcal{S}$ ends in S-LEQT or S-LEQF are analogous to S-PLUS. (Note that for the Preservation lemma, unlike for Progress, we do not need to use that the side conditions on S-LEQT and S-LEQF are exhaustive; but we do need to check that *both* of them reduce $\overline{n_0} \le \overline{n_1}$ to a well-typed boolean term.)

- Case $\mathcal{S} = \text{S-IFT}\dfrac{}{\mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2 \to t_1}$, so $t = \mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2$ and $t' = t_1$.

  The typing derivation for t must look like:

$$\mathcal{T} = \text{T-IF}\frac{\text{T-TRUE}\dfrac{}{[] \vdash \mathbf{true} : \mathbf{bool}} \qquad \overset{\mathcal{T}_1}{[] \vdash t_1 : \tau} \qquad \overset{\mathcal{T}_2}{[] \vdash t_2 : \tau}}{[] \vdash \mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2 : \tau}$$

  But then we can directly take $\mathcal{T}' = \mathcal{T}_1$ as the typing derivation for t'. The case where $\mathcal{S}$ ends in S-IFF is analogous.

- Case $\mathcal{S} = \text{S-FST}\dfrac{}{\mathbf{fst}((c_1, c_2)) \to c_1}$, so $t = \mathbf{fst}((c_1, c_2))$ and $t' = c_1$.

  Here we must have, for some τ_1 and τ_2:

$$\mathcal{T} = \text{T-FST}\dfrac{\text{T-PAIR}\dfrac{\overset{\mathcal{T}_1}{[] \vdash c_1 : \tau_1} \qquad \overset{\mathcal{T}_2}{[] \vdash c_2 : \tau_2}}{[] \vdash (c_1, c_2) : \tau_1 \times \tau_2}}{[] \vdash \mathbf{fst}((c_1, c_2)) : \tau_1},$$

  with $\tau = \tau_1$. Again, we can just take $\mathcal{T}' = \mathcal{T}_1$ as the typing derivation for t'. The case where $\mathcal{S}$ ends in S-SND is analogous.

- Case $\mathcal{S} = \text{S-APP}\dfrac{}{(\lambda x.\, t_0)\, c_2 \to t_0[c_2/x]}$, so $t = (\lambda x.\, t_0)\, c_2$ and $t' = t_0[c_2/x]$.

  The typing derivation for t must look like:

$$\mathcal{T} = \text{T-APP}\dfrac{\text{T-LAM}\dfrac{\overset{\mathcal{T}_0}{[x \mapsto \tau_1] \vdash t_0 : \tau_2}}{[] \vdash \lambda x.\, t_0 : \tau_1 \to \tau_2} \qquad \overset{\mathcal{T}_1}{[] \vdash c_2 : \tau_1}}{[] \vdash (\lambda x.\, t_0)\, c_2 : \tau_2},$$

  and $\tau = \tau_2$. By Lemma 4.10 on $\mathcal{T}_0$ and $\mathcal{T}_1$ (taking $\Gamma = []$ and $\tau' = \tau_1$), we directly get the required derivation $\mathcal{T}'$ of $[] \vdash t_0[c_2/x] : \tau$.

- (The cases for **let**-terms are left as an exercise.)

- Case $\mathcal{S} = \text{S-REC}\dfrac{}{\mathbf{rec}\ x.\, t_0 \to t_0[(\mathbf{rec}\ x.\, t_0)/x]}$, so $t = \mathbf{rec}\, x.\, t_0$ and $t' = t_0[(\mathbf{rec}\, x.\, t_0)/x]$.

  Here we must have:

$$\mathcal{T} = \text{T-REC}\dfrac{\overset{\mathcal{T}_0}{[x \mapsto \tau] \vdash t_0 : \tau}}{[] \vdash \mathbf{rec}\ x.\, t_0 : \tau}$$

  And so we get $\mathcal{T}'$ by Lemma 4.10 on $\mathcal{T}_0$ and $\mathcal{T}$, taking t' in the lemma as t itself.

■

Note that the term t' after reduction has the same type τ as the original t. This is not actually needed to show safety of evaluation (after all, if the new term is still well typed, even with a different type, it will not get stuck either), but it is used for the induction to go through in all the context-rule cases.

Theorem 4.12 (Type soundness) *If $[] \vdash t : \tau$ then t is safe.*

Proof. Suppose $t \to^* t'$ by some derivation $\mathcal{SS}$; we must show that t' is not stuck. The argument is by a simple induction on $\mathcal{SS}$: If the step sequence is of length zero, then $t' = t$, and so the typing of t guarantees that it is not stuck, by the Progress lemma. On the other hand, if $t \to t''$ and $t'' \to^* t'$, then by Preservation, we also have $[] \vdash t'' : \tau$; and hence by IH on the now shorter sequence $t'' \to^* t'$, we again get that t' is not stuck. ■

Although our type system evidently guarantees safety (which is, in general, a semantically undecidable property, because it talks about reducing a term for unboundedly

many steps), it is necessarily *conservative*: it may reject some dubious-looking terms that actually happen to be safe. For example,

$$\textbf{if true then } \overline{3} \textbf{ else } (\overline{5} + \textbf{true})$$

is clearly safe (it reduces to the canonical form $\overline{3}$ in one step), but it is nevertheless rejected by the type system. So is (**if true then** $\overline{5}$ **else true**) $+ \overline{3}$. However, in some type systems (notably, so-called *dependent types*, where the types of terms may depend on the *values* of their subterms), the latter term, or even both, might be well-typed of type **int**.

4.2.3 Church vs. Curry typing

The previous presentation is often known as the *Curry view* of typing: we define the operational semantics of terms first, and then impose a type system on them to guarantee some properties (in our case, safety). The problem of finding types for (a priori untyped) terms is then called *type inference*. In general, there can be many different type systems for the same language, and a given term may have zero, one, or many types, depending on which type system we employ.

An alternative approach is known as the *Church view*, in which the notion of types comes *before* terms, and we only ever assign meanings (i.e., operational semantics) to well-typed terms. The Church view is a natural extension of the approach we used for IMP, where we considered a term like $\overline{5} + \textbf{true}$ as inherently ill-formed, though in FUN it is rejected by a more refined system than a simple context-free grammar. In the Church view, every well-typed term (and there is no other kind!) has a unique type, which is often emphasized by annotating the places in the syntax that bind variables with *typing tags*, such as $\lambda x^{\textbf{int}}.\, x$ or $\lambda x\colon \textbf{int}.\, x$. In many cases, however, such tags can be *reconstructed* from how the term is introduced or used, in which case it is not necessary to write them out explicitly in the program. We will return to type inference/reconstruction later in the course.

4.3 Termination proofs

We now set out to show that any closed, well-typed term without recursion actually evaluates to a canonical from. While this may seem obvious at first (after all, infinite recursion would appear to be the only way to ever write a diverging term), on closer analysis it is actually much more subtle. For example, the term $\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ evidently steps to itself by S-App, and so can never be reduced to a canonical form. Fortunately, as argued earlier, it is not typable either, since it contains the untypable subterm $\lambda x.\, x\, x$.

The essence of the termination argument is to find a suitably strengthened induction hypothesis: we shall show not only that every well-typed term evaluates to a canonical form c, but also that this c satisfies an additional property, derived from its type. This proof method is an instance of a general technique called *logical relations*, or *Tait's method*, and represents yet another "proof skeleton" that we will show for FUN only, but that can be extended to richer languages. It turns out that the proof works a

little smoother in a big-step semantics, but since we have already shown the equivalence between the big-step and small-step semantics, the result of course transfers immediately.

We first define, by structural on induction τ, two predicates on (not necessarily well-typed) canonical forms c and general closed terms t, respectively:

$$\begin{aligned}
\models^{\mathrm{c}} c : \mathbf{int} &\iff \exists n.\, c = \overline{n} \\
\models^{\mathrm{c}} c : \mathbf{bool} &\iff c = \mathbf{true} \vee c = \mathbf{false} \\
\models^{\mathrm{c}} c : \tau_1 \times \tau_2 &\iff \exists c_1, c_2.\, c = (c_1, c_2) \wedge \models^{\mathrm{c}} c_1 : \tau_1 \wedge \models^{\mathrm{c}} c_2 : \tau_2 \\
\models^{\mathrm{c}} c : \tau_1 \to \tau_2 &\iff \exists x, t_0.\, c = \lambda x.\, t_0 \wedge \forall c'.\, \models^{\mathrm{c}} c' : \tau_1 \Rightarrow \models^{\mathrm{t}} t_0[c'/x] : \tau_2 \\
\models^{\mathrm{t}} t : \tau &\iff \exists c.\, t \downarrow c \wedge \models^{\mathrm{c}} c : \tau
\end{aligned}$$

Note that the predicate $\models^{\mathrm{c}} - : \tau$ for non-atomic types τ is defined in terms of the predicates for types strictly smaller than τ; and $\models^{\mathrm{t}} - : \tau$ can always just be replaced with its definition. Thus, there is no circularity.

The definition of the predicate for function types in particular requires a canonical form $\models^{\mathrm{c}} c : \tau_1 \to \tau_2$ to be an abstraction $c = \lambda x.\, t_0$ representing a *total* function (i.e., evaluation of its body must terminate successfully), but only for arguments that themselves satisfy the predicate at type τ_1. (Consider, e.g., $c = \lambda f.\, (\overline{0} \leq f\,\overline{3})$, with $\models^{\mathrm{c}} c : (\mathbf{int} \to \mathbf{int}) \to \mathbf{bool}$: the higher-order function c is only guaranteed to be total for arguments c' that are themselves total functions, $\models^{\mathrm{c}} c' : \mathbf{int} \to \mathbf{int}$.)

The predicates express a *semantic* notion of well-typedness, defined by how the term behaves, according to the operational semantics. It is not the same as *syntactic* typing, defined by inference rules. For example, for $c_1 = \lambda x.\, \mathbf{rec}\ y.\, y$ we have $[] \vdash c_1 : \mathbf{bool} \to \mathbf{int}$, but not $\models^{\mathrm{c}} c_1 : \mathbf{bool} \to \mathbf{int}$ (e.g., because $\models^{\mathrm{c}} \mathbf{true} : \mathbf{bool}$, but not $\models^{\mathrm{t}} (\mathbf{rec}\ y.\, y)[\mathbf{true}/x] : \mathbf{int}$, as that term evidently does not evaluate to a numeral). On the other hand, for $c_2 = \lambda x.\, \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \overline{3}\ \mathbf{else}\ (\overline{5} + x)$, we have $\models^{\mathrm{c}} c_2 : \mathbf{bool} \to \mathbf{int}$, but not $[] \vdash c_2 : \mathbf{bool} \to \mathbf{int}$ (because the **else**-branch is untypable in a typing context where x has type **bool**).

For a finite sequence of canonical forms $\vec{c} = (c^1, ..., c^k)$ and a corresponding sequence of distinct variables $\vec{x} = (x^1, ..., x^k)$, let us write $s = \vec{c}/\vec{x}$, and $t[s]$ for the iterated substitution $t[c^1/x^1] \cdots [c^k/x^k]$. (We write the indices as superscripts merely to avoid name clashes with unrelated numberings of terms and variables in the following.) Note that, since the c^i are closed, the order in which we perform the substitutions does not matter.

We say that such a substitution is (semantically) well typed according to a type context, if the canonical form that we substitute for each variable matches that variable's designated type:

$$\models^{\mathrm{c}} s : \Gamma \iff \forall x \in \mathrm{dom}\,\Gamma.\, \models^{\mathrm{c}} x[s] : \Gamma(x)$$

For open terms t, we can then define a *semantic* notion of well-typedness in a context:

$$\Gamma \models t : \tau \iff \forall s.\, \models^{\mathrm{c}} s : \Gamma \Rightarrow \models^{\mathrm{t}} t[s] : \tau$$

That is, the term t has semantic type τ in type context Γ if, after substituting all t's free variables with canonical forms semantically typable according to Γ, the resulting closed term evaluates to a canonical form of semantic type τ.

The termination property for closed terms will now follow from a more general result about *open* terms:

Theorem 4.13 (Termination) *If* $\Gamma \vdash t : \tau$ *without using rule* T-Rec, *then* $\Gamma \models t : \tau$.

Proof. Let $\mathcal{T}$ be a derivation of $\Gamma \vdash t : \tau$, and let s be such that, whenever $\Gamma(x) = \tau'$, then $\models^{\mathrm{c}} x[s] : \tau'$. We must show that $\models^{\mathrm{t}} t[s] : \tau$, i.e., that $t[s] \downarrow c$ for some c with $\models^{\mathrm{c}} c : \tau$. We proceed by induction on $\mathcal{T}$.

- Case $\mathcal{T} = \text{T-Var}\dfrac{}{\Gamma \vdash x : \tau}$, where $\Gamma(x) = \tau$.

  Then, by the assumption on s, we immediately get that $\models^{\mathrm{c}} x[s] : \tau$. And hence, by Lemma 4.3 (i.e., that the canonical form $c = x[s]$ evaluates to itself), also $\models^{\mathrm{t}} x[s] : \tau$.

- Case $\mathcal{T} = \text{T-Num}\dfrac{}{\Gamma \vdash \overline{n} : \mathbf{int}}$, so $\tau = \mathbf{int}$.

  Then, we evidently have $\overline{n}[s] \downarrow \overline{n}$, and $\models^{\mathrm{c}} \overline{n} : \mathbf{int}$, as required.

- The cases for T-True and T-False are analogous to T-Num.

- Case $\mathcal{T} = \text{T-Plus}\dfrac{\begin{array}{cc}\mathcal{T}_0 & \mathcal{T}_1 \\ \Gamma \vdash t_0 : \mathbf{int} & \Gamma \vdash t_1 : \mathbf{int}\end{array}}{\Gamma \vdash t_0 + t_1 : \mathbf{int}}$, so $t = t_0 + t_1$.

  By IH on $\mathcal{T}_0$, $\models^{\mathrm{t}} t_0[s] : \mathbf{int}$, so there is an $\mathcal{E}_0$ of $t_0[s] \downarrow c_0$ for some $\models^{\mathrm{c}} c_0 : \mathbf{int}$, which means that $c_0 = \overline{n_0}$ for some n_0. Analogously, by IH on $\mathcal{T}_1$, we get an $\mathcal{E}_1$ of $t_1[s] \downarrow \overline{n_1}$ for some n_1. But then we can use E-Plus on $\mathcal{E}_0$ and $\mathcal{E}_1$ to show that $t_0[s] + t_1[s] \downarrow \overline{n_0 + n_1}$; and since $t[s] = t_0[s] + t_1[s]$, and $\models^{\mathrm{c}} \overline{n_0 + n_1} : \mathbf{int}$, we are done.

- The case for T-Leq is analogous, only noting at the end that if $n_0 \leq n_1$, we can use use E-LeqT (because $\models^{\mathrm{c}} \mathbf{true} : \mathbf{bool}$), and similarly for $n_0 > n_1$.

- Case $\mathcal{T} = \text{T-If}\dfrac{\begin{array}{ccc}\mathcal{T}_0 & \mathcal{T}_1 & \mathcal{T}_2 \\ \Gamma \vdash t_0 : \mathbf{bool} & \Gamma \vdash t_1 : \tau & \Gamma \vdash t_2 : \tau\end{array}}{\Gamma \vdash \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$.

  By IH on $\mathcal{T}_0$, we get a derivation $\mathcal{E}_0$ of $t_0[s] \downarrow c_0$, where $\models^{\mathrm{c}} c_0 : \mathbf{bool}$. Suppose $c_0 = \mathbf{true}$; then by IH on $\mathcal{T}_1$, we get a derivation $\mathcal{E}_1$ of $t_1[s] \downarrow c$ for some $\models^{\mathrm{c}} c : \tau$; and by E-IfT on $\mathcal{E}_0$ and $\mathcal{E}_1$, we obtain $\mathbf{if}\ t_0[s]\ \mathbf{then}\ t_1[s]\ \mathbf{else}\ t_2[s] \downarrow c$, as required. The subcase for $c_0 = \mathbf{false}$ is analogous.

- Case $\mathcal{T} = \text{T-Pair}\dfrac{\begin{array}{cc}\mathcal{T}_1 & \mathcal{T}_2 \\ \Gamma \vdash t_1 : \tau_1 & \Gamma \vdash t_2 : \tau_2\end{array}}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}$, so $t = (t_1, t_2)$.

  By IH on $\mathcal{T}_1$, we get a derivation $\mathcal{E}_1$ of $t_1[s] \downarrow c_1$ with $\models^{\mathrm{c}} c_1 : \tau_1$; and similarly, by IH on $\mathcal{T}_2$, an $\mathcal{E}_2$ of $t_2[s] \downarrow c_2$ with $\models^{\mathrm{c}} c_2 : \tau_2$. But then by E-Pair on $\mathcal{E}_1$ and $\mathcal{E}_2$, we get $(t_1[s], t_2[s]) \downarrow (c_1, c_2)$, and $\models^{\mathrm{c}} (c_1, c_2) : \tau_1 \times \tau_2$ by definition.

- Case $\mathcal{T} = \text{T-Fst}\dfrac{\begin{array}{c}\mathcal{T}_0 \\ \Gamma \vdash t_0 : \tau_1 \times \tau_2\end{array}}{\Gamma \vdash \mathbf{fst}(t_0) : \tau_1}$, so $t = \mathbf{fst}(t_0)$ and $\tau = \tau_1$.

  By IH on $\mathcal{T}_0$, we get a derivation $\mathcal{E}_0$ of $t_0[s] \downarrow c$, for some $\models^{\mathrm{c}} c : \tau_1 \times \tau_2$, i.e., with $c = (c_1, c_2)$ for some c_1 and c_2, where in particular $\models^{\mathrm{c}} c_1 : \tau_1$. And then, by rule E-Fst on $\mathcal{E}_0$, we get $\mathbf{fst}(t_0[s]) \downarrow c_1$, as required. The case for T-Snd is analogous.

- Case $\mathcal{T} = \text{T-LAM}\dfrac{\begin{array}{c}\mathcal{T}_0\\ \Gamma[x \mapsto \tau_1] \vdash t_0 : \tau_2\end{array}}{\Gamma \vdash \lambda x.\, t_0 : \tau_1 \to \tau_2}$, so $t = \lambda x.\, t_0$ and $\tau = \tau_1 \to \tau_2$.

  Let $s' = \vec{c}'/\vec{x}'$ be the substitution $s = \vec{c}/\vec{x}$, but leaving x alone. More precisely, if $x \notin \vec{x}$, we just take $\vec{x}' = \vec{x}$ and $\vec{c}' = \vec{c}$; but if $x = x^i$ for some i, then $\vec{x}' = (x^1, ..., x^{i-1}, x^{i+1}, ..., x^k)$ and $\vec{c}' = (c^1, ..., c^{i-1}, c^{i+1}, ..., c^k)$. In either case, $x \notin \vec{x}'$, and $t[s] = (\lambda x.\, t_0)[s] = \lambda x.\, t_0[s']$.

  By E-LAM, we have a trivial derivation of $\lambda x.\, t_0[s'] \downarrow \lambda x.\, t_0[s']$, so it remains to show that $\models^{\text{c}} \lambda x.\, t_0[s'] : \tau_1 \to \tau_2$. Accordingly, let $\models^{\text{c}} c' : \tau_1$; we must show that $\models^{\text{t}} t_0[s'][c'/x] : \tau_2$. But since $x \notin \vec{x}'$, $t_0[s'][c'/x] = t_0[\vec{c}'/\vec{x}'][c'/x] = t_0[(\vec{c}', c')/(\vec{x}', x)]$. Let $s_1 = (\vec{c}', c')/(\vec{x}', x)$ and $\Gamma_1 = \Gamma[x \mapsto \tau_1]$; we can then check that $\models^{\text{c}} s_1 : \Gamma_1$: for any $y \in \operatorname{dom} \Gamma_1$, we must show $\models^{\text{c}} y[s_1] : \Gamma_1(y)$. If $y = x$ then $y[(\vec{c}', c')/(\vec{x}', x)] = c'$ and $\models^{\text{c}} c' : \Gamma[x \mapsto \tau_1](y)$; and if $y \neq x$, then $y[(\vec{c}', c')/(\vec{x}', x)] = y[\vec{c}'/\vec{x}'] = y[s]$, and $\models^{\text{c}} y[s] : \Gamma[x \mapsto \tau_1](y)$ by the assumption on s and Γ. Thus, we obtain the needed $\models^{\text{t}} t_0[s_1] : \tau_2$ directly by IH on the subderivation $\mathcal{T}_0$.

- Case $\mathcal{T} = \text{T-APP}\dfrac{\begin{array}{c}\mathcal{T}_1\\ \Gamma \vdash t_1 : \tau_1 \to \tau_2\end{array} \qquad \begin{array}{c}\mathcal{T}_2\\ \Gamma \vdash t_2 : \tau_1\end{array}}{\Gamma \vdash t_1\, t_2 : \tau_2}$, so $t = t_1\, t_2$ and $\tau = \tau_2$.

  By IH on $\mathcal{T}_1$, we get a derivation $\mathcal{E}_1$ of $t_1[s] \downarrow c_1$, with $\models^{\text{c}} c_1 : \tau_1 \to \tau$; and by IH on $\mathcal{T}_2$, an $\mathcal{E}_2$ of $t_2[s] \downarrow c_2$, where $\models^{\text{c}} c_2 : \tau_1$. But by the semantic typing of c_1, we must have $c_1 = \lambda x.\, t_0$ for some x and t_0; and moreover, taking c' as c_2, we get $\models^{\text{t}} t_0[c_2/x] : \tau_2$, which means that there is an $\mathcal{E}_3$ of $t_0[c_2/x] \downarrow c$ for some $\models^{\text{c}} c : \tau$. We can then put together the three evaluation derivations with E-APP to get the required derivation of $t_1[s]\, t_2[s] \downarrow c$.

- (The case for T-LET is left as an exercise.)

- (There is no case for T-REC!) ∎

We then get, as a special case of the theorem, that any well-typed closed term t (i.e., such that $[] \vdash t : \tau$ for some τ), not containing **rec**, must evaluate to a canonical form in the big-step semantics, and by Theorem 4.2, also in the small-step semantics. In particular, we get an alternative proof that well-typed terms are safe, though that version, unlike the Progress+Preservation approach, only works for terms without recursion.

4.4 Exercises

In some of the exercises, we consider an extension of the FUN syntax with the following term form:

$$t ::= \cdots \mid \mathbf{min}\ x \geq t_0.\, t_1$$

where x is considered bound in t_1. Informally, it evaluates to a numeral representing the smallest integer n, greater than or equal to the value of t_0, such that t_1 with x replaced by $\overline{n}$ evaluates to **true**. For example, $\mathbf{min}\ x \geq \overline{0}.\, \overline{9} \leq x + x$ would evaluate to $\overline{5}$. The candidates for n are tried sequentially, starting from the value n_0 of t_0, then n_0+1, n_0+2,

etc. If no solution exists, or if t_1 diverges for some candidate before a solution is found, the whole term diverges.

Formally, we define the big-step semantics as follows:

$$\text{E-MinT} : \frac{t_0 \downarrow \overline{n_0} \qquad t_1[\overline{n_0}/x] \downarrow \mathbf{true}}{\mathbf{min}\ x \geq t_0.\, t_1 \downarrow \overline{n_0}}$$

$$\text{E-MinF} : \frac{t_0 \downarrow \overline{n_0} \qquad t_1[\overline{n_0}/x] \downarrow \mathbf{false} \qquad \mathbf{min}\ x \geq \overline{n_0 + 1}.\, t_1 \downarrow c}{\mathbf{min}\ x \geq t_0.\, t_1 \downarrow c}$$

And the small-step one:

$$\text{S-Min1} : \frac{t_0 \to t_0'}{\mathbf{min}\ x \geq t_0.\, t_1 \to \mathbf{min}\ x \geq t_0'.\, t_1}$$

$$\text{S-Min} : \frac{}{\mathbf{min}\ x \geq \overline{n_0}.\, t_1 \to \mathbf{if}\ t_1[\overline{n_0}/x]\ \mathbf{then}\ \overline{n_0}\ \mathbf{else}\ \mathbf{min}\ x \geq \overline{n_0 + 1}.\, t_1}$$

Moreover, we define the typing rule for **min** as follows:

$$\text{T-Min} : \frac{\Gamma \vdash t_0 : \mathbf{int} \qquad \Gamma[x \mapsto \mathbf{int}] \vdash t_1 : \mathbf{bool}}{\Gamma \vdash \mathbf{min}\ x \geq t_0.\, t_1 : \mathbf{int}}$$

4.1. Prove Theorem 4.2 and Lemma 4.4 for the cases relating to **min**.

4.2. For any boolean term t, potentially containing an integer variable x, let the term $MIN(x, t)$ of type $\mathbf{int} \to \mathbf{int}$ be defined by:

$$MIN(x, t) \equiv \mathbf{rec}\ f.\, \lambda x.\, \mathbf{if}\ t\ \mathbf{then}\ x\ \mathbf{else}\ f\,(x + \overline{1})$$

(where we pick some $f \notin FV(t)$). Show the following, by induction on derivations:

$$\mathbf{min}\ x \geq t_0.\, t_1 \downarrow c \Rightarrow MIN(x, t_1)\, t_0 \downarrow c$$

(The other direction also holds, but you are not asked to prove that.)

4.3. Extend the proofs of Lemmas 4.8 (Progress), 4.10 (Substitution), 4.11 (Preservation), and Theorem 4.13 (Termination) with the cases for **let**-terms. (Start by writing out $FV(t)$ and $t[t'/x]$ from Section 4.1.2 for the case where t is a **let**-term.)

4.4. Extend the proofs of Lemmas 4.8 (Progress) and 4.11 (Preservation) with the cases for **min**-terms. (Though the Substitution lemma in principle also needs to be extended to show Preservation, you are not required to show that extension, which is quite standard.)