Notes on Programming Language Theory

Richard Willie

July 27, 2025

Preface

These notes are a loose amalgamation of ideas, concepts, and explanations drawn from various books, papers, and personal reflections. They were originally meant for my own understanding and organization of thoughts, and as such, they may be unpolished, incomplete, or even occasionally incorrect.

I share them in the hope that they may serve as a useful reference, but they should not be treated as a primary source of learning. Readers are strongly encouraged to consult original texts and authoritative resources for a more rigorous and accurate treatment of the topics discussed.

Use these notes as a companion to your studies, not as a substitute for the depth and clarity provided by well-established literature.

Contents

1 Typed Arithmetic Expressions

§1.1 Types

The syntax for arithmetic expressions:

`t` ::=		term
	`true`	constant true
	`false`	constant false
	`if t then t else t`	conditional
	`0`	constant zero
	`succ t`	successor
	`pred t`	predecessor
	`iszero t`	zero test

Evaluating a term can either result in a value...

`v` ::=		value
	`true`	true value
	`false`	false value
	`nv`	numeric value

`nv` ::=		numeric value
	`0`	zero value
	`succ nv`	successor value

or else get *stuck* at some stage, by reaching a term for which no evaluation rule applies.

Stuck terms correspond to meaningless or erroneous programs. We would therefore like to be able to tell, without actually evaluating a term, that its evaluation will definitely *not* get stuck. To do this, we introduce two types, `Nat` and `Bool`, to distinguish between terms whose result will be a numeric value and terms whose result will be a boolean.

§1.2 The Typing Relation

The typing relation for arithmetic expressions, written `t : T`, is defined by a set of inference rules assigning types to terms, summarized as follows.

New syntactic forms:

`T` ::=		type
	`Bool`	type of booleans
	`Nat`	type of natural numbers

New typing rules:

$$\texttt{true} : \texttt{Bool} \quad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \quad \text{(T-False)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 : \texttt{T}} \quad \text{(T-If)}$$

$$\texttt{0} : \texttt{Nat} \quad \text{(T-Zero)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ t}_1 : \texttt{Nat}} \quad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred t}_1 : \texttt{Nat}} \quad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero t}_1 : \texttt{Bool}} \quad \text{(T-IsZero)}$$

Definition 1.2.1

Formally, the **typing relation** for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the rules above. A term `t` is **well typed** if there is some `T` such that $\texttt{t} : \texttt{T}$.

When reasoning about the typing relation, we will often make statements like "If a term of the form $\texttt{succ t}_1$ has any type at all, then it has type `Nat`." The following lemma gives us a compendium of basic statements of this form, each following immediately from the shape of the corresponding typing rule.

Lemma 1.2.1 (Inversion lemma for typed arithmetic expressions)

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$ and $\texttt{t}_2 : \texttt{R}$ and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ t}_1 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Nat}$ and $\texttt{R} = \texttt{Nat}$.
6. If $\texttt{pred t}_1 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Nat}$ and $\texttt{R} = \texttt{Nat}$.
7. If $\texttt{iszero t}_1 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Nat}$ and $\texttt{R} = \texttt{Bool}$.

Proof. Immediate from the definition of the typing relation. □

The inversion lemma is sometimes called the *generation lemma* for the typing relation, since, given a valid typing statement, it shows how a proof of this statement could have been generated. The inversion lemma leads directly to a recursive algorithm for calculating the types of terms, since it tells us, for a term of each syntactic form, how to calculate its type (if it has one) from the types of its subterms.

Theorem 1.2.2 (Uniqueness of types for typed arithmetic expressions)

Each term `t` has at most one type. That is, if `t` is typable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules above.

Proof. Straightforward structural induction on `t`, using the appropriate clause of the inversion lemma (plus the induction hypothesis) for each case. □

§1.3 Safety = Progress + Preservation

The most basic property of this type system or any other is *safety* (or *soundness*), i.e. well-typed terms do not go wrong. We have already chosen how to formalize what it means for a term to go wrong: it means reaching a stuck state that is not designated as a final value but where evaluation rules do not tell us what do next. What we want to know, then, is that well-typed terms do not get stuck. We show this in two steps, commonly known as the *progress* and *preservation* theorems.

Theorem 1.3.1 (Progress)

A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules). Formally, suppose `t` is a well-typed term (that is, $\texttt{t} : \texttt{T}$ for some `T`). Then either `t` is a value, or else there is some $\texttt{t}'$ with $\texttt{t} \to \texttt{t}'$.

Proof. By induction on a derivation of $\texttt{t} : \texttt{T}$. The T-True, T-False, and T-Zero cases are immediate, since `t` in these cases is a value. For the other caes, we argue as follows.

Case T-If: $\texttt{t} = \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$, where $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{T}$, and $\texttt{t}_3 : \texttt{T}$. By the induction hypothesis, either $\texttt{t}_1$ is a value or else there is some $\texttt{t}_1'$ such that $\texttt{t}_1 \to \texttt{t}_1'$. If $\texttt{t}_1$ is a value, then it must be either `true` or `false`, in which case either E-IfTrue or E-IfFalse applies to `t`. On the other hand, if $\texttt{t}_1 \to \texttt{t}_1'$, then by E-If, we have $\texttt{t} \to \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$.

Case T-Succ: $\texttt{t} = \texttt{succ } \texttt{t}_1$, where $\texttt{t}_1 : \texttt{Nat}$. By the induction hypothesis, either $\texttt{t}_1$ is a value or else there is some $\texttt{t}_1'$ such that $\texttt{t}_1 \to \texttt{t}_1'$. If $\texttt{t}_1$ is a value, then it must be a numeric value, in which case so is `t`. On the other hand, if $\texttt{t}_1 \to \texttt{t}_1'$, then by E-Succ, we have $\texttt{t} \to \texttt{succ } \texttt{t}_1'$.

Case T-Pred: $\texttt{t} = \texttt{pred } \texttt{t}_1$, where $\texttt{t}_1 : \texttt{Nat}$. By the induction hypothesis, either $\texttt{t}_1$ is a value or else there is some $\texttt{t}_1'$ such that $\texttt{t}_1 \to \texttt{t}_1'$. If $\texttt{t}_1$ is a value, then it must be a numeric value, either `0` or $\texttt{succ } \texttt{nv}_1$ for some $\texttt{nv}_1$, in which case either E-PredZero or E-PredSucc applies to `t`. On the other hand, if $\texttt{t}_1 \to \texttt{t}_1'$, then by E-Pred, we have $\texttt{t} \to \texttt{pred } \texttt{t}_1'$.

Case T-IsZero: $\texttt{t} = \texttt{iszero } \texttt{t}_1$, where $\texttt{t}_1 : \texttt{Nat}$. Similar. □

Theorem 1.3.2 (Preservation)

If a well-typed term takes a step of evaluation, then the resulting term is also well typed. Formally, if $\texttt{t} : \texttt{T}$ and $\texttt{t} \to \texttt{t}'$, then $\texttt{t}' : \texttt{T}$.

Proof. By induction on a derivation of $\mathtt{t} : \mathtt{T}$. At each step of the induction, we assume that the desired property holds for all subderivations and proceed by case analysis on the final rule of the derivation.

Case T-TRUE: $\mathtt{t} = \mathtt{true}$ and $\mathtt{T} = \mathtt{Bool}$. Vacuously true.

Case T-FALSE: $\mathtt{t} = \mathtt{false}$ and $\mathtt{T} = \mathtt{Bool}$. Vacuously true.

Case T-IF: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$, where $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{T}$, and $\mathtt{t}_3 : \mathtt{T}$. There are three rules by which $\mathtt{t} \to \mathtt{t}'$ can be derived: E-IFTRUE, E-IFFALSE, and E-IF. We consider each case separately (omitting E-IFFALSE).

- Subcase E-IFTRUE: $\mathtt{t}_1 = \mathtt{true}$ and $\mathtt{t}' = \mathtt{t}_2$. If $\mathtt{t} \to \mathtt{t}'$ is derived using E-IFTRUE, then from the form of this rule we see that $\mathtt{t}_1$ must be $\mathtt{true}$ and the resulting term $\mathtt{t}'$ is the second subexpression $\mathtt{t}_2$. This means we are done, since we know (by the assumption of the T-IF case) that $\mathtt{t}_2 : \mathtt{T}$, which is what we need.
- Subcase E-IF: $\mathtt{t}_1 \to \mathtt{t}_1'$ and $\mathtt{t}' = \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$. From the assumptions of the T-IF case, we have a subderivation of the original typing derivation whose conclusion is $\mathtt{t}_1 : \mathtt{Bool}$. We can apply the induction hypothesis to this subderivation, obtaining $\mathtt{t}_1' : \mathtt{Bool}$. Combining this with the facts (from the assumptions of the T-IF case) that $\mathtt{t}_2 : \mathtt{T}$ and $\mathtt{t}_3 : \mathtt{T}$, we can apply rule T-IF to conclude that $\mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}$.

Case T-ZERO: $\mathtt{t} = \mathtt{0}$ and $\mathtt{T} = \mathtt{Nat}$. Vacuously true.

Case T-SUCC: $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1$, where $\mathtt{T} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$. There is just one rule, E-SUCC, that can be used to derive $\mathtt{t} \to \mathtt{t}'$. The form of this rule tells us that $\mathtt{t}_1 \to \mathtt{t}_1'$. Since we also know $\mathtt{t}_1 : \mathtt{Nat}$, we can apply the induction hypothesis to obtain $\mathtt{t}_1' : \mathtt{Nat}$, from which we obtain $\mathtt{succ}\ \mathtt{t}_1' : \mathtt{Nat}$ by applying rule T-SUCC.

Case T-PRED: $\mathtt{t} = \mathtt{pred}\ \mathtt{t}_1$, where $\mathtt{T} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$. Similar.

Case T-ISZERO: $\mathtt{t} = \mathtt{iszero}\ \mathtt{t}_1$, where $\mathtt{T} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$. Similar. □

2 Simply Typed Lambda Calculus

§2.1 The Typing Relation

Since terms may contain nested λ-abstractions, we will need, in general, to talk about several assumptions. This changes the typing relation from a two-place relation, $\mathtt{t} : \mathtt{T}$, to a three-place relation, $\Gamma \vdash \mathtt{t} : \mathtt{T}$, where Γ is a set of assumptions about the types of the free variables in $\mathtt{t}$.

Definition 2.1.1

Formally, a **typing context** (or a **type environment**) Γ is a sequence of variables and their types, and the comma operator extends Γ by adding a new binding on the right.

The rule of typing abstraction has the general form:

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \text{ (T-Abs)}$$

where the premise adds one more assumption to those in the conclusion.

The typing rules for variables:

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \text{ (T-Var)}$$

The typing rules for applications:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\, \mathtt{t}_2 : \mathtt{T}_{12}} \text{ (T-App)}$$

For completeness, we summarize the syntax and evaluation rules for simply-typed lambda calculus as follows.

Syntax:

$\mathtt{t}$ ::=		term
	$\mathtt{x}$	variable
	$\lambda \mathtt{x}:\mathtt{T}.\mathtt{t}$	abstraction
	$\mathtt{t}\,\mathtt{t}$	application
$\mathtt{v}$ ::=		value
	$\lambda \mathtt{x}:\mathtt{T}.\mathtt{t}$	abstraction value
$\mathtt{T}$ ::=		type
	$\mathtt{T} \to \mathtt{T}$	type of functions

Evaluation:

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1\,\mathtt{t}_2 \to \mathtt{t}_1'\,\mathtt{t}_2}\ (\text{E-App1})$$

$$\frac{\mathtt{t}_2 \to \mathtt{t}_2'}{\mathtt{v}_1\,\mathtt{t}_2 \to \mathtt{v}_1\,\mathtt{t}_2'}\ (\text{E-App2})$$

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_2)\,\mathtt{v}_2 \to [\mathtt{x}\mapsto\mathtt{v}_2]\mathtt{t}_2\ (\text{E-AppAbs})$$

§2.2 Properties of Typing

As in Chapter 1, we need to develop a few basic lemmas before we can prove type safety. Most of these are similar to what we saw before, we just need to add contexts to the typing relation and add clauses to each proof for λ- abstractions, applications, and variables. The only significant new requirement is a substitution lemma for the typing relation (Lemma 2.2.6).

First off, an inversion lemma records a collection of observations about how typing derivations are built: the clause for each syntactic form tells us "if a term of this form is well typed, then its subterms must have types of these forms..."

Lemma 2.2.1 (Inversion lemma for simply typed lambda calculus)

1. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\mathtt{x}{:}\mathtt{R} \in \Gamma$.
2. If $\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{R}_2$.
3. If $\Gamma \vdash \mathtt{t}_1\,\mathtt{t}_2 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{R}$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$ for some type $\mathtt{T}_{11}$.

Proof. Immediate from the definition of the typing relation. □

In Section 2.1, we chose an explicitly typed presentation of the calculus to simplify the job of typechecking. This involved adding type annotations to bound variables in function abstractions, but nowhere else. In what sense is this "enough"? One answer is provided by the "uniqueness of types" theorem, which tells us that well-typed terms are in one-to-one correspondence with their typing derivations: the typing derivation can be recovered uniquely from the term (and, of course, vice versa). In fact, the correspondence is so straightforward that, in a sense, there is little difference between the term and the derivation.

Theorem 2.2.2 (Uniqueness of types for simply typed lambda calculus)

In a given typing context Γ, a term $\mathtt{t}$ (with free variables all in the domain of Γ) has at most one type. That is, if a term is typable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules that generate the typing relation.

Proof. Straightforward. □

We can prove a progress theorem analogous to Theorem 1.3.1. The statement of the theorem needs one small change: we are interested only in *closed* terms, with no free variables. For open terms, the progress theorem actually fails: a term like $\mathtt{x}$ is a normal

form, but not a value. However, this failure does not represent a defect in the language, since complete programs, which are the terms we actually care about evaluating, are always closed.

Theorem 2.2.3 (Progress)
Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\vdash \mathtt{t} : \mathtt{T}$ for some $\mathtt{T}$). Then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \to \mathtt{t}'$.

Proof. Straightforward induction on typing derivations. The variable case cannot occur (because $\mathtt{t}$ is closed). The abstraction case is immediate, since abstractions are values. The only interesting case is the one for application, where $\mathtt{t} = \mathtt{t}_1\,\mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-App1 applies to $\mathtt{t}$. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-App2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then $\mathtt{t}_1$ must be an abstraction of the form $\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12}$, and so rule E-AppAbs applies to $\mathtt{t}$. □

Our next job is to prove that evaluation preserves types. We begin by stating a couple of "structural lemmas" for the typing relation. These are not particularly interesting in themselves, but will permit us to perform some useful manipulations of typing derivations in later proofs.

Lemma 2.2.4 (Permutation)
If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former.

Proof. Straightforward induction on typing derivations. □

Lemma 2.2.5 (Weakening)
If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathrm{dom}(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$ for any type $\mathtt{S}$. Moreover, the latter derivation has the same depth as the former.

Proof. Straightforward induction on typing derivations. □

Using these technical lemmas, we can prove a crucial property of the typing relation: that well-typedness is preserved when variables are substituted with terms of appropriate types.

Lemma 2.2.6 (Preservation of types under substitution)
If $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$.

Proof. By induction on a derivation of the statement $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$. For a given derivation, we proceed by cases on the final typing rule used in the proof. The most interesting cases are the ones for variables and abstractions.

Case T-Var: $\mathtt{t} = \mathtt{z}$ with $\mathtt{z{:}T} \in (\Gamma, \mathtt{x{:}S})$. There are two subcases to consider, depending on whether $\mathtt{z}$ is $\mathtt{x}$ or another variable. If $\mathtt{z} = \mathtt{x}$, then $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t} = \mathtt{s}$. The required result is then $\Gamma \vdash \mathtt{s} : \mathtt{S}$, which is among the assumptions of the lemma. Otherwise, $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t} = \mathtt{z}$, and the desired result is immediate.

Case T-Abs: $\mathtt{t} = \lambda\mathtt{y{:}T_2}.\mathtt{t_1}$, $\mathtt{T} = \mathtt{T_2}\to\mathtt{T_1}$, and $\Gamma, \mathtt{x{:}S}, \mathtt{y{:}T_2} \vdash \mathtt{t_1} : \mathtt{T_1}$. We may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin FV(\mathtt{s})$. Using permutation on the given subderivation, we obtain $\Gamma, \mathtt{y{:}T_2}, \mathtt{x{:}S} \vdash \mathtt{t_1} : \mathtt{T_1}$. Using weakening on the other given derivation ($\Gamma \vdash s : S$), we obtain $\Gamma, \mathtt{y{:}T_2} \vdash \mathtt{s} : \mathtt{S}$. Now, by the induction hypothesis, we have $\Gamma, \mathtt{y{:}T_2} \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_1} : \mathtt{T_1}$. Finally, we can apply the rule T-Abs to conclude that $\Gamma \vdash \lambda\mathtt{y{:}T_2}.[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$, which is the desired result since $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t} = \lambda\mathtt{y{:}T_2}.[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_1}$ and $\mathtt{T} = \mathtt{T_2}\to\mathtt{T_1}$.

Case T-App: $\mathtt{t} = \mathtt{t_1}\,\mathtt{t_2}$, $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$, $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t_2} : \mathtt{T_2}$, and $\mathtt{T} = \mathtt{T_1}$. By the induction hypothesis, we have $\Gamma, \mathtt{x{:}S} \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$ and $\Gamma, \mathtt{x{:}S} \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_2} : \mathtt{T_2}$. By the rule T-App, we can conclude that $\Gamma \vdash ([\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_1})\,([\mathtt{x}\mapsto\mathtt{s}]\mathtt{t_2}) : \mathtt{T}$. That is, $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}](\mathtt{t_1}\,\mathtt{t_2}) : \mathtt{T}$, which is the desired result. □

Theorem 2.2.7 (Preservation)
If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \to \mathtt{t'}$, then $\Gamma \vdash \mathtt{t'} : \mathtt{T}$.

Proof. By induction on a derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. that if $\Gamma \vdash \mathtt{s} : \mathtt{S}$ and $\mathtt{s} \to \mathtt{s'}$, then $\Gamma \vdash \mathtt{s'} : \mathtt{S}$, whenever $\Gamma \vdash \mathtt{s} : \mathtt{S}$ is proved by a subderivation of the present one) and proceed by case analysis on the last rule used in the derivation.

Case T-Var: $\mathtt{t} = \mathtt{x}$ with $\mathtt{x{:}T} \in \Gamma$. Can't happen.

Case T-Abs: $\mathtt{t} = \lambda\mathtt{x{:}T_1}.\mathtt{t_2}$. Can't happen.

Case T-App: $\mathtt{t} = \mathtt{t_1}\,\mathtt{t_2}$, $\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}}\to\mathtt{T_{12}}$, $\Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}$, and $\mathtt{T} = \mathtt{T_{12}}$. There are three rules by which $\mathtt{t} \to \mathtt{t'}$ can be derived: E-App1, E-App2, and E-AppAbs. We consider each case separately.

- Subcase E-App1: $\mathtt{t_1} \to \mathtt{t_1'}$ and $\mathtt{t'} = \mathtt{t_1'}\,\mathtt{t_2}$. From the assumptions of the T-App case, we have a subderivation of the original typing derivation whose conclusion is $\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}}\to\mathtt{T_{12}}$. We can apply the induction hypothesis to this subderivation, obtaining $\Gamma \vdash \mathtt{t_1'} : \mathtt{T_{11}}\to\mathtt{T_{12}}$. Combining this with the facts (also from the assumptions of the T-App case) that $\Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}$, we can apply rule T-App to conclude that $\Gamma \vdash \mathtt{t'} : \mathtt{T}$.
- Subcase E-App2: $\mathtt{t_1} = \mathtt{v_1}$, $\mathtt{t_2} \to \mathtt{t_2'}$, and $\mathtt{t'} = \mathtt{v_1}\mathtt{t_2'}$. Similar.
- Subcase E-AppAbs: $\mathtt{t_1} = \lambda\mathtt{x{:}T_{11}}.\mathtt{t_{12}}$, $\mathtt{t_2} = \mathtt{v_2}$, and $\mathtt{t'} = [\mathtt{x}\mapsto\mathtt{v_2}]\mathtt{t_{12}}$. We can deconstruct the typing derivation for $\lambda\mathtt{x{:}T_{11}}.\mathtt{t_{12}}$, yielding $\Gamma, \mathtt{x{:}T_{11}} \vdash \mathtt{t_{12}} : \mathtt{T_{12}}$. From this and the substitution lemma (Lemma 2.2.6), we obtain $\Gamma \vdash \mathtt{t'} : \mathtt{T_{12}}$.

□

3 Simple Extensions

§3.1 Base Types

Every programming language provides a variety of *base types* plus appropriate primitive operations to manipulate these values. For theoretical purposes, it is often useful to abstract away from the details of particular base types and their operations, and instead simply suppose that our language comes equipped with some set $\mathcal{A}$ of *uninterpreted* base types, with no primitive operations at all. This is accomplished simply by including the elements of $\mathcal{A}$ in the set of types, as shown below.

New syntactic forms:

T	::=	...	type
		A	base type

§3.2 The Unit Type

Another useful base type, found especially in languages in the ML family, is the singleton type `Unit`, described below.

New syntactic forms:

t	::=	...	term
		`unit`	constant unit
v	::=	...	value
		`unit`	constant unit
T	::=	...	type
		`Unit`	unit type

New typing rules:

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \quad \text{(T-Unit)}$$

In contrast to the uninterpreted base types, the unit type is interpreted in the simplest possible way: we explicitly introduce a single element, the term constant `unit` and a typing rule making `unit` an element of `Unit`. We also add `unit` to the set of possible result values of computations.

§3.3 Derived Forms: Sequencing and Wildcards

In languages with side effects, it is often useful to evaluate two or more expressions in sequence. The sequencing notation $\texttt{t}_1\texttt{;t}_2$ has the effect of evaluating $\texttt{t}_1$, throwing away its trivial result, and going on to evaluate $\texttt{t}_2$.

There are actually two different ways to formalize sequencing. One is to follow the same pattern we haved used for syntactic forms: add $\texttt{t}_1\texttt{;t}_2$ as a new alternative in the syntax of terms, and then add two evaluation rules

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{t}_1\texttt{;t}_2 \to \texttt{t}_1'\texttt{;t}_2} \text{ (E-SEQ)}$$

$$\texttt{unit;t}_2 \to \texttt{t}_2 \text{ (E-SEQNEXT)}$$

and a typing rule

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Unit} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{t}_1\texttt{;t}_2 : \texttt{T}_2} \text{ (T-SEQ)}$$

capturing the intended behavior of `;`.

An alternative way of formalizing sequencing is simply to regard $\texttt{t}_1\texttt{;t}_2$ as an abbreviation for the term $(\lambda\texttt{x:Unit.t}_2)\texttt{t}_1$, where the variable `x` is chosen fresh, i.e. different from all the free variables of $\texttt{t}_2$.

It is intuitively fairly clear that these two presentations of sequencing add up to the same thing as far as the programmer is concerned: the high-level typing and evaluation rules for sequencing can be derived from the abbreviation of $\texttt{t}_1\texttt{;t}_2$ as $(\lambda\texttt{x.Unit.t}_2)\texttt{t}_1$. This intuitive correspondence is captured more formally by arguing that typing and evaluation both "commute" with the expansion of the abbreviation.

New derived forms:

$$\texttt{t}_1\texttt{;t}_2 \stackrel{\text{def}}{=} (\lambda\texttt{x:Unit.t}_2)\,\texttt{t}_1 \text{ where } \texttt{x} \notin FV(\texttt{t}_2)$$

§3.4 Let Bindings

When writing a complex expression, it is often useful to give names to some of its subexpressions. Most languages provide one or more ways of doing this. In ML, for example, we write $\texttt{let x=t}_1\texttt{ in t}_2$ to mean "evaluate expression $\texttt{t}_1$ and bind the name `x` to the resulting value, while evaluating $\texttt{t}_2$."

New syntactic forms:

t ::=	...		term
	let x=t in t		let binding

New evaluation rules:

$$\texttt{let x=v}_1\texttt{ in t}_2 \to [\texttt{x}\mapsto\texttt{v}_1]\texttt{t}_2 \text{ (E-LETV)}$$

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{let x=t}_1\texttt{ in t}_2 \to \texttt{let x=t}_1'\texttt{ in t}_2} \text{ (E-LET)}$$

New typing rules:

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x=t}_1\texttt{ in t}_2 : \texttt{T}_2} \text{ (T-LET)}$$

Our `let`-binder follows ML's in choosing a call-by-value evaluation order, where the `let`-bound term must be fully evaluated before evaluation of the `let`-body can begin. The typing rule T-Let tells us that the type of a `let` can be calculated 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.

Can `let` also be defined as a derived form? Yes, as Landin showed, but the details are slightly more subtle than what we did for sequencing an ascripiton. Naively, it is clear that we use a combination of abstraction and application to achieve the effect of a `let`-binding:

$$\texttt{let x=t}_1\ \texttt{in t}_2 \stackrel{\text{def}}{=} (\lambda \texttt{x:T}_1.\texttt{t}_2)\,\texttt{t}_1$$

But notice that the right-hand side of this abbreviation includes the type annotation $\texttt{T}_1$, which does not appear on the left-hand side. That is, if we imagine a derived forms as being desugared during the parsing phase of some compiler, then we need to ask how the parser is supposed to know that it should generate $\texttt{T}_1$ as the type annotation on the λ in the desugared internal language term.

The answer, of course, is that this information comes from the typechecker! We discover the needed type annotation simply by calculating the type of $\texttt{t}_1$. More formally, what this tells us is that the `let` constructor is a slightly different sort of derived form than the ones we have seen so far: we should regard it as a transformation on the *typing derivations* that maps a derivation involving `let`:

$$\frac{\dfrac{\vdots}{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1} \qquad \dfrac{\vdots}{\Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}}{\Gamma \vdash \texttt{let x=t}_1\ \texttt{in t}_2 : \texttt{T}_2}\ (\text{T-Let})$$

to one using abstraction and application:

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}}{\Gamma \vdash \lambda \texttt{x:T}_1.\texttt{t}_2 : \texttt{T}_1 \to \texttt{T}_2}\ (\text{T-Abs}) \qquad \dfrac{\vdots}{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1}}{\Gamma \vdash (\lambda \texttt{x:T}_1.\texttt{t}_2)\,\texttt{t}_1 : \texttt{T}_2}\ (\text{T-App})$$

Thus, `let` is a little less derived than the other derived forms we have seen: we can derive its evalution behavior by desugaring it, but its typing behavior must be built into the internal language.

§3.5 General Recursion

Another facility found in most programming languages is the ability to define recursive functions. Such functions can be defined with the aid of the **fixed-point combinator**:

$$\texttt{fix} = \lambda \texttt{f}.(\lambda \texttt{x}.\texttt{f}(\lambda \texttt{y}.\texttt{x x y}))\,(\lambda \texttt{x}.\texttt{f}(\lambda \texttt{y}.\texttt{x x y}))$$

Suppose we want to write a recursive function definition of the form `h =` ⟨body containing `h`⟩. The intention is that the recursive definition should be "unrolled" at the point where it occurs; for example, the definition of `factorial` would intuitively be:

```
if n = 0 then 1
else n * (if (n-1)=0 then 1
          else (n-1) * (if (n-2)=0 then 1
                        else (n-2) * ...))
```

This effect can be achieved using the `fix` combinator by first defining `g = λf.`⟨body containing `f`⟩ and then `h = fix g`. For example, we can define the factorial function by

```
g = λfct.(λn.(if n = 0 then 1 else n * (fct (n-1))))
factorial = fix g
```

Recursive functions can be defined in a typed setting in a similar way. However, there is one important difference: `fix` itself cannot be defined in the simply typed lambda calculus. Indeed, we will see in Chapter 4 that expression that can lead to non-terminating computations cannot be typed using only simple types. So, instead of defining `fix` as a term in the language, we simply add it as a new primitive, with evaluatiion rules mimicking the behavior of the untyped `fix` combinator and a typing rule that captures its intended uses.

New syntactic forms:

$\mathtt{t}$	::=	...	term
		$\mathtt{fix\ t}$	fixed point of $\mathtt{t}$

New evaluation rules:

$$\mathtt{fix\ (\lambda x{:}T_1.t_2)} \to \mathtt{[x \mapsto (fix\ (\lambda x{:}T_1.t_2))]t_2} \quad \text{(E-FixBeta)}$$

$$\frac{\mathtt{t_1} \to \mathtt{t_1'}}{\mathtt{fix\ t_1} \to \mathtt{fix\ t_1'}} \quad \text{(E-Fix)}$$

New typing rules:

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \to \mathtt{T_1}}{\Gamma \vdash \mathtt{fix\ t_1} : \mathtt{T_1}} \quad \text{(T-Fix)}$$

New derived forms:

$$\mathtt{letrec\ x{:}T_1{=}t_1\ in\ t_2} \stackrel{\text{def}}{=} \mathtt{let\ x{=}fix(\lambda x{:}T_1.t_1)\ in\ t_2}$$

The simply typed lambda calculus with numbers and `fix` has long been the favorite experimental subject of programming language researchers, since it is the simplest language in which a range of subtle semantics phenomena such as *full abstraction* arise. It is often called PCF.

4 Normalization

In this chapter, we consider another fundamental property of the pure simply typed lambda calculus: the fact that the evaluation of a well-typed program is guaranteed to halt in a finite number of steps. That is, every well-typed term is *normalizable*.

§4.1 Normalization for Simple Types

The calculus we shall consider here is the simply typed lambda calculus over a single base type `A`. Normalization for this calculus is not entirely trivial to prove, since each reduction of a term can duplicate redexes in subterms.

Exercise 4.1.1. Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term?

We fail because of the substitution that occurs during β-reduction (E-AppAbs). When an application of the form $(\lambda \mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\,\mathtt{v}_2$ reduces to $[\mathtt{x}\mapsto\mathtt{v}_2]\mathtt{t}_{12}$, the resulting term may be larger than the original term. Consider this example:

- Original term: $(\lambda \mathtt{x}:\mathtt{A}.\mathtt{x}\,\mathtt{x}\,\mathtt{x})(\lambda \mathtt{y}:\mathtt{A}.\mathtt{y}\,\mathtt{y})$ (11 nodes)
- After reduction: $(\lambda \mathtt{y}:\mathtt{A}.\mathtt{y}\,\mathtt{y})(\lambda \mathtt{y}:\mathtt{A}.\mathtt{y}\,\mathtt{y})(\lambda \mathtt{y}:\mathtt{A}.\mathtt{y}\,\mathtt{y})$ (14 nodes)

The key issue here (as in many proofs by induction) is finding a strong enough induction hypothesis.

Definition 4.1.1

We define, for each type $\mathtt{T}$, a set $\mathcal{R}_{\mathtt{T}}$ of closed terms of type $\mathtt{T}$ as follows:

- For the base type $\mathtt{A}$, $\mathcal{R}_{\mathtt{A}}(\mathtt{t})$ holds if and only if $\mathtt{t}$ halts.
- For function types, $\mathcal{R}_{\mathtt{T}_1\to\mathtt{T}_2}(\mathtt{t})$ holds if and only if $\mathtt{t}$ halts and for every $\mathtt{s}$ such that $\mathcal{R}_{\mathtt{T}_1}(\mathtt{s})$ holds, we also have $\mathcal{R}_{\mathtt{T}_2}(\mathtt{t}\,\mathtt{s})$.

This gives us the strengthened induction hypothesis that we need. Our primary goal is to show that all programs (i.e. all closed terms of base type) halt. But closed terms of base type can contain subterms of functional type, so we need to know something about these as well. Moreover, it is not enough to know that these subterms halt, because the application of a normalized function to a normalized argument involves a substitution, which may enable more evaluation steps. So we need a stronger condition for terms of functional type: not only should they halt themselves, but, when applied to halting arguments, they should yield halting results.

The form of Definition 4.1.1 is characteristic of the *logical relations* proof technique. (Since we are just dealing with unary relations here, we should more properly say *logical predicates*.) If we want to prove some property $\mathcal{P}$ of all closed terms of type `A`, we proceed by proving, by induction on types, that all terms of type `A` *possess* property $\mathcal{P}$, all terms of type $\mathtt{A}\to\mathtt{A}$ *preserve* property $\mathcal{P}$, all terms of type $(\mathtt{A}\to\mathtt{A})\to(\mathtt{A}\to\mathtt{A})$ *preserve*

the property of preserving property $\mathcal{P}$, and so on. We do this by defining a family of predicates, indexed by types. For the base type $\mathtt{A}$, the predicate is just $\mathcal{P}$. For functional types, it says that the function should map values satisfying the predicate at the input type to values satisfying the predicate at the output type.

We use this definition to carry out the proof the proof of normalization in two steps. First, we observe that every element of the set $\mathcal{R}_{\mathtt{T}}$ is normalizable. Then we show that every well-typed term of type $\mathtt{T}$ is an element of $\mathcal{R}_{\mathtt{T}}$.

The first step is immediate from the definition of $\mathcal{R}_{\mathtt{T}}$.

Lemma 4.1.2

If $\mathcal{R}_{\mathtt{T}}(\mathtt{t})$, then $\mathtt{t}$ halts.

The second step is broken down into two lemmas. First, we remark that membership in $\mathcal{R}_{\mathtt{T}}$ is invariant under evaluation.

Lemma 4.1.3

If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \to \mathtt{t}'$, then $\mathcal{R}_{\mathtt{T}}(\mathtt{t})$ if and only if $\mathcal{R}_{\mathtt{T}}(\mathtt{t}')$.

Proof. By induction on the structure of the type $\mathtt{T}$. Note, first, that it is clear that $\mathtt{t}$ halts if and only if $\mathtt{t}'$ does.

- If $\mathtt{T} = \mathtt{A}$, then there is nothing more to show.
- If $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$ for some $\mathtt{T}_1$ and $\mathtt{T}_2$. There are two directions:

  ($\Rightarrow$) $\mathcal{R}_{\mathtt{T}}(\mathtt{t})$, i.e. $\mathtt{t}$ halts and $\mathcal{R}_{\mathtt{T}_2}(\mathtt{t}\,\mathtt{s})$ whenever $\mathcal{R}_{\mathtt{T}_1}(\mathtt{s})$ for some arbitrary $\vdash \mathtt{s} : \mathtt{T}_1$. Inductive hypothesis: If $\vdash \mathtt{t} : \mathtt{T}_2$ and $\mathtt{t} \to \mathtt{t}'$, then $\mathcal{R}_{\mathtt{T}_2}(\mathtt{t})$ iff $\mathcal{R}_{\mathtt{T}_2}(\mathtt{t}')$. Our premise tells us that $\mathtt{t} \to \mathtt{t}'$, so we have $\mathtt{t}\,\mathtt{s} \to \mathtt{t}'\,\mathtt{s}$. Applying the inductive hypothesis gives us $\mathcal{R}_{\mathtt{T}_2}(\mathtt{t}'\,\mathtt{s})$. Since this holds for arbitrary $\mathtt{s}$, the definition of $\mathcal{R}_{\mathtt{T}}$ gives us $\mathcal{R}_{\mathtt{T}}(\mathtt{t}')$.

  ($\Leftarrow$) Analogous.

□

Next, we want to show that every term of type $\mathtt{T}$ belongs to $\mathcal{R}_{\mathtt{T}}$. Here, the induction will be on typing derivations (it would be surprising to see a proof about well-typed terms that did not somewhere involve induction on typing derivations!). The only technical difficulty here is in dealing with the λ-abstraction case. Since we are arguing by induction, the demonstration that a term $\lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2$ belongs to $\mathcal{R}_{\mathtt{T}_1 \to \mathtt{T}_2}$ should involve applying the induction hypothesis to show that $\mathtt{t}_2$ belongs to $\mathcal{R}_{\mathtt{T}_2}$. But $\mathcal{R}_{\mathtt{T}_2}$ is defined to be a set of *closed* terms, while $\mathtt{t}_2$ may contain free $\mathtt{x}$, so this does not make sense.

This problem is resolved by using a standard trick to suitably generalize the induction hypothesis: instead of proving a statement involving a closed term, we generalize it to cover all closed *instances* of an open term $\mathtt{t}$.

Lemma 4.1.4

If $\mathtt{x}_1{:}\mathtt{T}_1, \ldots, \mathtt{x}_n{:}\mathtt{T}_n \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{v}_1, \ldots, \mathtt{v}_n$ are closed values of types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ with $\mathcal{R}_{\mathtt{T}_i}(\mathtt{v}_i)$ for each i, then $\mathcal{R}_{\mathtt{T}}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t})$.

Proof. By induction on a derivation of $\mathtt{x}_1{:}\mathtt{T}_1, \ldots, \mathtt{x}_n{:}\mathtt{T}_n \vdash \mathtt{t} : \mathtt{T}$. (The most interesting case is the one for abstraction.)

Case T-Var: $\mathtt{t} = \mathtt{x}_i$ and $\mathtt{T} = \mathtt{T}_i$ for some i. Immediate.

Case T-Abs: $\mathtt{t} = \lambda \mathtt{x}{:}\mathtt{S}_1 . \mathtt{s}_2$, $\mathtt{x}_1{:}\mathtt{T}_1, \ldots, \mathtt{x}_n{:}\mathtt{T}_n, \mathtt{x}{:}\mathtt{S}_1 \vdash \mathtt{s}_2 : \mathtt{S}_2$, $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$. Obviously, $[\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]t$ evaluates to a value, since it is a value already. What remains to show is that $\mathcal{R}_{\mathtt{S}_2}(([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t})\ \mathtt{s})$ for any $\mathtt{s} : \mathtt{S}_1$ such that $\mathcal{R}_{\mathtt{S}_1}(\mathtt{s})$. Suppose $\mathtt{s}$ is such a term. By Lemma 4.1.2, we have $\mathtt{s} \to^* \mathtt{v}$ for some $\mathtt{v}$. By Lemma 4.1.3, $\mathcal{R}_{\mathtt{S}_1}(v)$. Now, by the inductive hypothesis, $\mathcal{R}_{\mathtt{S}_2}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}_2)$. Applying the evaluation rule E-AppAbs gives us

$$(\lambda \mathtt{x}{:}\mathtt{S}_1 . [\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}_2)\ \mathtt{s} \to^* [\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}_2$$

from which Lemma 4.1.3 gives us

$$\mathcal{R}_{\mathtt{S}_2}((\lambda \mathtt{x}{:}\mathtt{S}_1 . [\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}_2)\ \mathtt{s}).$$

That is,

$$\mathcal{R}_{\mathtt{S}_2}(([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,[\mathtt{x} \mapsto \mathtt{v}]\,(\lambda \mathtt{x}{:}\mathtt{S}_1 . \mathtt{s}_2))\ \mathtt{s}).$$

Since $\mathtt{s}$ was chosen arbitrarily, the definition of $\mathcal{R}_{\mathtt{S}_1 \to \mathcal{S}_2}$ gives us

$$\mathcal{R}_{\mathtt{S}_1 \to \mathtt{S}_2}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,(\lambda \mathtt{x}{:}\mathtt{S}_1 . \mathtt{s}_2)).$$

Case T-App: $\mathtt{t} = \mathtt{t}_1\,\mathtt{t}_2$, $\mathtt{x}_1{:}\mathtt{T}_1, \ldots, \mathtt{x}_n{:}\mathtt{T}_n \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$, $\mathtt{x}_1{:}\mathtt{T}_1, \ldots, \mathtt{x}_n{:}\mathtt{T}_n \vdash \mathtt{t}_2 : \mathtt{T}_{11}$, $\mathtt{T} = \mathtt{T}_{12}$. The inductive hypothesis gives us $\mathcal{R}_{\mathtt{T}_{11} \to \mathtt{T}_{12}}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t}_1)$ and $\mathcal{R}_{\mathtt{T}_{11}}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t}_2)$. By the definition of $\mathcal{R}_{\mathtt{T}_{11} \to \mathtt{T}_{12}}$, we have

$$\mathcal{R}_{\mathtt{T}_{12}}(([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t}_1)([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\mathtt{t}_2)).$$

That is,

$$\mathcal{R}_{\mathtt{T}_{12}}([\mathtt{x}_1 \mapsto \mathtt{v}_1] \cdots [\mathtt{x}_n \mapsto \mathtt{v}_n]\,(\mathtt{t}_1\,\mathtt{t}_2)).$$

□

Theorem 4.1.5 (Normalization)

If $\vdash \mathtt{t} : \mathtt{T}$, then $\mathtt{t}$ is normalizable.

Proof. We obtain the normalization property as a corollary, simply by taking the term $\mathtt{t}$ to be closed in Lemma 4.1.4 and then recalling that all elements of $\mathcal{R}_{\mathtt{T}}$ are normalizable by Lemma 4.1.2. □