Type Systems

Benjamin C. Pierce
`bcpierce@cis.upenn.edu`

Working draft of November 7, 1999

This is preliminary draft of a book in progress. Comments, suggestions, and corrections are welcome.

Contents

1 Introduction **6**
1.1 Applications of Type Systems . . . 6
1.2 Varieties of Type Systems . . . 7
1.3 History . . . 7
1.4 Further Reading . . . 9

2 Mathematical Preliminaries **10**
2.1 Sets and Relations . . . 10
2.2 Induction . . . 10
2.3 Term Rewriting . . . 11

I Untyped Systems **12**

3 Arithmetic Expressions **13**
3.1 Basics . . . 13
3.1.1 Syntax . . . 14
3.1.2 Operational Semantics . . . 14
3.2 Formalities . . . 14
3.2.1 Syntax . . . 14
3.2.2 Evaluation . . . 18
3.2.3 Properties . . . 19
3.3 An ML Implementation . . . 19
3.3.1 Syntax . . . 20
3.3.2 Evaluation . . . 20
3.4 Summary . . . 21

4 The Lambda-Calculus **22**
4.1 Basics . . . 23
4.1.1 Syntax . . . 24
4.1.2 Operational Semantics . . . 25
4.2 Programming in the Lambda-Calculus . . . 25

4.3 Formalities 30
4.3.1 Syntax 30
4.3.2 Substitution 31
4.3.3 Operational Semantics 34
4.3.4 Summary 34
4.4 Further Reading 35

II First-Order Systems **36**

5 Typed Arithmetic Expressions **37**
5.1 Syntax 37
5.2 The Typing Relation 37
5.3 Properties of Typing and Reduction 38
5.3.1 Typing Derivations 38
5.3.2 Typechecking 39
5.3.3 Type Soundness 39
5.4 Implementation 40

6 Simple Types **42**
6.1 Syntax 42
6.2 The Typing Relation 43
6.3 Summary 45
6.4 Properties of Typing and Reduction 46
6.4.1 Typechecking 47
6.4.2 Typing and Substitution 49
6.4.3 Type Soundness 49
6.5 Implementation 49
6.6 Further Reading 51

7 Extensions **52**
7.1 Base Types 52
7.2 Unit type 52
7.3 Let bindings 53
7.4 Records and Tuples 53
7.5 Variants 56
7.6 General recursion 56
7.7 Lists 56
7.8 Lazy records and let-bindings 56

8 Computational Effects **57**
8.1 References 57
8.1.1 Summary 57
8.1.2 Properties 61

8.2 Further Reading 62

9 Type Equivalence **63**

10 Subtyping **64**
10.1 The Subtype Relation 65
10.1.1 Variance 66
10.1.2 Summary 67
10.2 Extensions 69
10.2.1 Primitive Subtyping 69
10.2.2 The Bottom Type 69
10.3 Metatheory of Subtyping 69
10.3.1 Algorithmic Subtyping 69
10.3.2 Minimal Typing 70
10.4 Implementation 74
10.5 Meets and Joins 75

11 Imperative Objects **77**
11.1 Objects 77
11.2 Object Generators 79
11.3 Subtyping 80
11.4 Basic classes 80
11.5 Extending the Internal State 82
11.6 Classes with "Self" 83

III Second-Order Systems **86**

12 Universal Types **87**
12.1 Motivation 87
12.2 Varieties of Polymorphism 88
12.3 Definitions 89
12.4 Examples 91
12.4.1 Warm-ups 92
12.4.2 Polymorphic Lists 93
12.4.3 Impredicative Encodings 93
12.5 Metatheory 96
12.5.1 Subject Reduction 96
12.5.2 Strong Normalization 96
12.5.3 Erasure and Typeability 96
12.5.4 Type Reconstruction 98
12.6 Implementation 98
12.6.1 Nameless Representation of Types 98
12.6.2 ML Code 98

12.7 Further Reading . . . 99

13 Existential Types **100**
13.1 Motivation . . . 100
13.2 Data Abstraction with Existentials . . . 105
13.2.1 Abstract Data Types . . . 105
13.2.2 Existential Objects . . . 108
13.2.3 Objects vs. ADTs . . . 110
13.3 Encoding Existentials . . . 110
13.4 Implementation . . . 112
13.5 Historical Notes . . . 112

14 Bounded Quantification **113**
14.1 Motivation . . . 113
14.2 Definitions . . . 115
14.2.1 Kernel $F_{<:}$ . . . 116
14.2.2 Full $F_{<:}$ . . . 117
14.3 Examples . . . 118
14.3.1 Encoding Products . . . 118
14.3.2 Encoding Records . . . 119
14.3.3 Church Encodings with Subtyping . . . 120
14.4 Metatheory . . . 122
14.4.1 Subject Reduction . . . 122
14.4.2 Typing . . . 126
14.4.3 Subtyping in $F^k_{<:}$ . . . 129
14.4.4 Subtyping in $F^f_{<:}$ . . . 132
14.5 Undecidability of Subtyping in Full $F_{<:}$ . . . 134
14.6 Bounded Existential Types . . . 138
14.7 Historical Notes and Further Reading . . . 139

Appendices 141

A Solutions to Selected Exercises **141**

B Summary of Notation **144**
B.1 Metavariable Conventions . . . 144
B.2 Type System Names . . . 144

C Running the Checkers **146**
C.1 Preparing Your Input . . . 146
C.2 Ascii Equivalents . . . 146
C.3 Running the Checker . . . 147
C.4 Compiling the Checkers . . . 147

C.5 Objective Caml 147
Bibliography **148**
Index **156**

Chapter 1

Introduction

Proofs of programs are too boring for the social process of mathematics to work.

—DeMillo, Lipton, and Perlis

...So don't rely on social processes for verification.
—David Dill

A **type system** is a formal method for classifying the phrases of a programming language according to the "shapes" of the results they return or the computations they engender.

1.1 Applications of Type Systems

Type systems are not just about detecting silly mistakes like `2+true`: they can be used to enforce programmer-defined abstractions as well as the abstractions embodied by the built-in elements of the language—to protect parts of the program from each other, as well as protecting the machine from the program.

Type systems are being applied across a broad range of areas in computer science...

- In programming in the large (module systems, interface definition languages, etc.)
- In compiling and optimization (static analyses, typed intermediate languages, typed assembly languages, etc.)
- In "self-certification" of untrusted code (so-called "proof-carrying code" [NL96, Nec97, NL98])
- In security

- In theorem proving
- In databases
- In linguistics (categorial grammar [Ben95, vBM97, etc.])
- In Y2K conversion tools

1.2 Varieties of Type Systems

A useful—though rough—distinction divides the world of programming languages into two parts:

- Untyped — programs simply execute flat out; there is no attempt to check "consistency of shapes"
- Typed — some attempt is made, either at compile time or at run-time, to check shape-consistency

Among typed languages, we can break things down further:

	Statically checked	Dynamically checked
Strongly typed	ML, Haskell, Pascal (almost), Java (almost)	Lisp, Scheme
Weakly typed	C, C++	Perl

1.3 History

The following table presents a (rough and incomplete) chronology of some important high points in the history of the area. Ideas coming more or less directly from logic are italicized, to give a sense of the importance of this field's contributions to the type systems used in computer science.

late 1800s	*Origins of formal logic*	[Frege]
early 1900s	*Formalization of mathematics*	[WR25]
'30s	*Untyped lambda-calculus*	[Chu41]
'40s	*Simply typed lambda-calculus*	[Chu40, CF58]
'50s	Fortran	[Bac81]
'50s	Algol	[N^+63]
'60s	*Automath project*	[dB80]
'60s	Simula	[BDMN79]
'70s	*Martin-Löf type theory*	[Mar73, Mar82, SNP90]
'60s	*Curry-Howard isomorphism*	[How80]

'70s	*System F*, F^{ω}	[Gir72]
'70s	polymorphic lambda-calculus	[Rey74]
'70s	CLU	[LAB+81]
'70s	polymorphic type inference	[Mil78, DM82]
'70s	Logic of Computable Functions	[GMW79]
'70s	ML	[GMW79]
'70s	*intersection types*	[CDC78, CDCS79, Pot80]
'80s	NuPRL project	[Con86]
'80s	subtyping	[Rey80, Car84, Mit84a]
'80s	ADTs as existential types	[MP88]
'80s	*calculus of constructions*	[Coq85, CH88]
'80s	*linear logic*	[Gir87, GLT89]
'80s	bounded quantification	[CW85, CG92, CMMS94]
'80s	*Edinburgh Logical Framework*	[HHP92]
'80s	Forsythe	[Rey88]
'80s	*pure type systems*	[Bar92]
'80s	dependent types and modularity	[Mac86]
'80s	Quest	[Car91]
'80s	*Extended Calculus of Constructions*	[Luo90]
'80s	row variables and extensible records	[Wan87, Rém89, CM91]
'90s	higher-order subtyping	[Car90, CL91, PT94]
'90s	typed intermediate languages	[TMC+96]
'90s	Object Calculus	[AC96]
'90s	translucent types and modarity	[HL94, Ler94]
'90s	typed assembly language	[MWCG98]

In computer science, the earliest type systems, beginning in the 1950s (e.g., FORTRAN), were used to improve efficiency of numerical calculations by distinguishing between natural-number-valued variables and arithmetic expressions and real-valued ones, allowing the compiler to use different representations and generate appropriate machine instructions for arithmetic operations. In the late '50s and early '60s (e.g., ALGOL), the classification was extended to structured data (arrays of records, etc.) and higher-order functions. Beginning in the '70s, these early foundations have been extended in many directions...

- **parametric polymorphism** allows a single term to be used with many different types (e.g., the same sorting routine might be used to sort lists of natural numbers, lists of reals, lists of records, etc.), encouraging code reuse;

- **module systems** support programming in the large by providing a framework for defining (and automatically checking) interfaces between the parts of a large software system;

- **subtyping** and **object types** address the special needs of object-oriented programming styles;

- connections are being developed between the type systems of programming languages, the **specification languages** used in program verification, and the **formal logics** used in theorem proving.

All of these (among many others) are still areas of active research.

1.4 Further Reading

- The handbook articles by Cardelli [Car96] and Mitchell [Mit90] offer quick introductions to the area.
- Mitchell's textbook on programming languages [Mit96] covers basic lambda calculus, a number of type systems, and semantics quite thoroughly. It's a bit light on bounded quantification and doesn't get to some of the interesting modern stuff.
- Abadi and Cardelli's *A Theory of Objects* [AC96]
- Girard's *Proofs and Types* [GLT89] treats logical aspects of type systems (the Curry-Howard isomorphism) very thoroughly. It also has a description of System F from the horse's mouth, and an appendix introducing linear logic.

If you want a single book (besides the one you're holding), I'd recommend [Mit96]. If you want two, I'd add [GLT89].

Chapter 2

Mathematical Preliminaries

Before getting started, we need to establish some common notation and state a few basic mathematical facts. Most readers should be able to skim this chapter and refer back to it as necessary.

2.1 Sets and Relations

2.2 Induction

2.2.1 Definition: A partially ordered set S is said to be **well founded** if it contains no infinite decreasing chains—that is, if there is no infinite sequence $s_1, s_2, s_3, \ldots$ of elements of S such that each s_{i+1} is strictly less than s_i. □

2.2.2 Theorem [Principle of well-founded induction]: Suppose that the set S is well founded and that P is some predicate on the elements of S. If we can show, for each $s : S$, that $(\forall s' < s.\ P(s'))$ implies $P(s)$, then we may conclude that $P(s)$ holds for every $s : S$. □

2.2.3 Corollary [Principle of induction on the natural numbers]: Suppose that P is some predicate on the natural numbers. If we can show, for each m, that $(\forall i < m.\ P(i))$ implies $P(m)$, then we may conclude that $P(n)$ holds for every n. □

Proof: The set of natural numbers is well founded. □

2.2.4 Corollary [Principle of lexicographic induction]: Define the following "dictionary ordering" on pairs of natural numbers: $(m, n) < (m', n')$ iff $m < m'$ or $m = m'$ and $n < n'$.

Now, suppose that P is some predicate on pairs of natural numbers. If we can show, for each (m, n), that $(\forall (m', n') < (m, n).\ P(m', n'))$ implies $P(m, n)$, then we may conclude that $P(m, n)$ holds for every pair (m, n).

(A similar principle holds for lexicographically ordered triples, quadruples, etc.) □

Proof: The lexicographic ordering on pairs of numbers is well founded. □

2.3 Term Rewriting

Part I

Untyped Systems

Chapter 3

Arithmetic Expressions

3.1 Basics

We begin with a very simple language for calculating with numbers and booleans.

```
   true;
▶ true

   if false then true else false;
▶ false

   0;
▶ 0

   succ (succ (succ 0));
▶ 3

   succ (pred 0);
▶ 1

   iszero (pred (succ 0));
▶ true

   iszero (succ (succ 0));
▶ false
```

Throughout the book, the symbol ▶ will be used to display the results of evaluating examples. You can think of the lines marked with ▶ as the responses from an interactive interpreter when presented with the preceding inputs.

For brevity, the examples use standard arabic numerals as shorthand for nested applications of `succ` to `0`, writing `succ(succ(succ(0)))` as `3`.

3.1.1 Syntax

The syntax of arithmetic expressions comprises several kinds of **terms**. The constants `true`, `false`, and `0` are terms. If `t` is a term, then so are `succ t`, `pred t`, and `iszero t`. Finally, if $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$ are terms, then so is $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$. These forms are summarized in the following **abstract grammar**:

`t`	::=	`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*

3.1.2 Operational Semantics

3.2 Formalities

3.2.1 Syntax

3.2.1.1 Definition [Terms]: The set of terms is the smallest set $\mathcal{T}$ such that

1. $\{\mathtt{true}, \mathtt{false}, \mathtt{0}\} \subseteq \mathcal{T}$;
2. if $\mathtt{t}_1 \in \mathcal{T}$, then $\{\mathtt{succ}\ \mathtt{t}_1, \mathtt{pred}\ \mathtt{t}_1, \mathtt{iszero}\ \mathtt{t}_1\} \subseteq \mathcal{T}$;
3. if $\mathtt{t}_1 \in \mathcal{T}$, $\mathtt{t}_2 \in \mathcal{T}$, and $\mathtt{t}_3 \in \mathcal{T}$, then $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \in \mathcal{T}$.

These three clauses capture exactly what is meant by the productions in the more concise and readable "abstract grammar" notation that we used above. □

Definition 3.2.1.1 is an example of an **inductive definition**. Since inductive definitions are ubiquitous in the study of programming languages, it is worth pausing for a moment to examine this one in detail. Here is an alternative definition of the same set, in a more concrete style.

3.2.1.2 Definition [Terms, more concretely]: For each natural number i, define a set $\mathcal{S}_i$ as follows:

$$
\begin{array}{lcll}
\mathcal{S}_0 & = & \emptyset & \\
\mathcal{S}_{i+1} & = & & \{\mathtt{true}, \mathtt{false}, \mathtt{0}\} \\
 & & \cup & \{\mathtt{succ}\ \mathtt{t}_1, \mathtt{pred}\ \mathtt{t}_1, \mathtt{iszero}\ \mathtt{t}_1 \mid \mathtt{t}_1 \in \mathcal{S}_i\} \\
 & & \cup & \{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \mid \mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3 \in \mathcal{S}_i\}.
\end{array}
$$

Finally, let

$$\mathcal{S} \quad = \quad \bigcup_i \mathcal{S}_i.$$

That is, $\mathcal{S}_0$ is empty; $\mathcal{S}_1$ contains just the constants; $\mathcal{T}_2$ contains the constants plus the phrases that can be built with constants and just one `succ`, `pred`, `iszero`, or `if`; $\mathcal{S}_3$ contains these plus all phrases that can be built using `succ`, `pred`, `iszero`, and `if` on phrases in $\mathcal{S}_2$; and so on. $\mathcal{S}$ collects together all the phrases that can be built in this way—i.e., all phrases built by some finite number of applications and abstractions, beginning with just variables. □

3.2.1.3 Exercise [Quick check]: List the elements of $\mathcal{S}_3$. □

Let us check that the two definitions of terms actually define the same set. We'll do the proof in quite a bit of detail, to show how all the pieces fit together.

3.2.1.4 Fact: $\mathcal{T} = \mathcal{S}$. □

Proof: $\mathcal{T}$ was defined as the smallest set satisfying certain conditions. So it suffices to show (a) that $\mathcal{S}$ satisfies these conditions, and (b) that any set satisfying the conditions has $\mathcal{S}$ as a subset (i.e., that $\mathcal{S}$ is the *smallest* set satisfying the conditions).

For part (a), we must check that each of the three conditions in Definition 3.2.1.1 holds of $\mathcal{S}$. First, since $\mathcal{S}_1 = \{\mathtt{true}, \mathtt{false}, \mathtt{0}\}$ and $\mathcal{S}_1 \subseteq \bigcup_i \mathcal{S}_i$, it is clear that the constants are in $\mathcal{S}$. Second, if $\mathtt{t}_1 \in \mathcal{S}$, then (since $\mathcal{S} = \bigcup_i \mathcal{S}_i$) there must be some i such that $\mathtt{t}_i \in \mathcal{S}_i$. But then, by the definition of $\mathcal{S}_{i+1}$, we must have $\mathtt{succ}\ \mathtt{t}_1 \in \mathcal{S}_{i+1}$, hence $\mathtt{succ}\ \mathtt{t}_1 \in \mathcal{S}$; similarly, we see that $\mathtt{pred}\ \mathtt{t}_1 \in \mathcal{S}$ and $\mathtt{iszero}\ \mathtt{t}_1 \in \mathcal{S}$. Third, if $\mathtt{t}_1 \in \mathcal{S}$, $\mathtt{t}_2 \in \mathcal{S}$, and $\mathtt{t}_3 \in \mathcal{S}$, then $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \in \mathcal{S}$, by a similar argument.

For part (b), suppose that some set $\mathcal{S}'$ satisfies the three conditions in Definition 3.2.1.1. We will argue, by induction on n, that every $\mathcal{S}_n \subseteq \mathcal{S}'$, from which it will clearly follow that $\mathcal{S} \subseteq \mathcal{S}'$. Suppose that $\mathcal{S}_m \subseteq \mathcal{S}'$ for all $m < n$; we must then show that $\mathcal{S}_n \subseteq \mathcal{S}'$. Since the definition of $\mathcal{S}_n$ has two clauses (for $n = 0$ and $n = i + 1$), there are two cases to consider.

- If $n = 0$, then $\mathcal{S}_n = \emptyset$. But $\emptyset \subseteq \mathcal{S}'$ trivially.
- Otherwise, $n = i + 1$ for some i. Let $\mathtt{t}$ be some element of $\mathcal{S}_{i+1}$. Since $\mathcal{S}_{i+1}$ is defined as the union of three smaller sets, $\mathtt{t}$ must come from one of these sets, and there are three possibilities to consider:
  1. $\mathtt{t}$ is a constant, hence $\mathtt{t} \in \mathcal{S}'$ by condition (1).
  2. $\mathtt{t}$ has the form $\mathtt{succ}\ \mathtt{t}_1$, $\mathtt{pred}\ \mathtt{t}_1$, or $\mathtt{iszero}\ \mathtt{t}_1$, for some $\mathtt{t}_1 \in \mathcal{S}_i$. But then, by the induction hypothesis, $\mathtt{t}_1 \in \mathcal{S}'$, and so, by condition (2), $\mathtt{t} \in \mathcal{S}'$.

3. $\mathtt{t}$ has the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$, for some $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3 \in \mathcal{S}_i$. Again, by the induction hypothesis, $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$ are all in $\mathcal{S}'$, and hence, by condition (3), so is $\mathtt{t}$.

Thus, we have shown that each $\mathcal{S}_i \subseteq \mathcal{S}'$. By the definition of $\mathcal{S}$ as the union of all the $\mathcal{S}_i$, this gives $\mathcal{S} \subseteq \mathcal{S}'$, completing the argument. □

The explicit characterization of $\mathcal{T}$ justifies an important principle for reasoning about its elements. If $\mathtt{t} \in \mathtt{T}$, then one of three things must be true about $\mathtt{t}$—either

1. $\mathtt{t}$ is a constant, or
2. $\mathtt{t}$ has the form $\mathtt{succ}\ \mathtt{t}_1$, $\mathtt{pred}\ \mathtt{t}_1$, or $\mathtt{iszero}\ \mathtt{t}_1$ for some *smaller* term $\mathtt{t}_1$, or
3. $\mathtt{t}$ has the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ for some *smaller* terms $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$.

We can put this observation to work in two ways: we can give *inductive definitions* of functions over the set of terms, and we can give *inductive proofs* of properties of terms. For example, here is a simple inductive definition of a function mapping each term $\mathtt{t}$ to the set of constants used in $\mathtt{t}$.

3.2.1.5 Definition: The set of constants appearing in a term $\mathtt{t}$, written $\mathit{Consts}(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl}
\mathit{Consts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{Consts}(\mathtt{iszero}) & = & \{\mathtt{iszero}\} \\
\mathit{Consts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_1)
\end{array}$$ □

Another property of terms that can be calculated by an inductive defintion is their size.

3.2.1.6 Definition: The **size** of a term $\mathtt{t}$, written $\mathit{size}(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl}
\mathit{size}(\mathtt{true}) & = & 1 \\
\mathit{size}(\mathtt{false}) & = & 1 \\
\mathit{size}(\mathtt{iszero}) & = & 1 \\
\mathit{size}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_1) + 1
\end{array}$$

(That is, the size of $\mathtt{t}$ is the number of nodes in its **abstract syntax tree**). Similarly, the **depth** of a term $\mathtt{t}$, written $depth(\mathtt{t})$, is defined as follows:

$$
\begin{array}{lcl}
depth(\mathtt{true}) & = & 1 \\
depth(\mathtt{false}) & = & 1 \\
depth(\mathtt{iszero}) & = & 1 \\
depth(\mathtt{succ\ t_1}) & = & depth(\mathtt{t_1}) + 1 \\
depth(\mathtt{pred\ t_1}) & = & depth(\mathtt{t_1}) + 1 \\
depth(\mathtt{iszero\ t_1}) & = & depth(\mathtt{t_1}) + 1 \\
depth(\mathtt{if\ t_1\ then\ t_2\ else\ t_3}) & = & \max(depth(\mathtt{t_1}), depth(\mathtt{t_1}), depth(\mathtt{t_1})) + 1
\end{array}
$$

Equivalently, $depth(\mathtt{t})$, is the smallest i such that $\mathtt{t} \in \mathcal{S}_i$. □

Here is an inductive proof of a simple fact relating the number of constants in a term to its size.

3.2.1.7 Lemma: The number of distinct constants in a term $\mathtt{t}$ is always smaller than the size of $\mathtt{t}$ (in symbols: $|Consts(\mathtt{t})| \leq size(\mathtt{t})$). □

(This property in itself is entirely obvious, of course. What's interesting is the form of the inductive proof, which we'll see repeated many times as we go along.)

Proof: By induction on the size of $\mathtt{t}$. That is, assuming the desired property for all terms smaller than $\mathtt{t}$, we must prove it for $\mathtt{t}$ itself; if we can do this, we may conclude that the property holds for all $\mathtt{t}$. There are three cases to consider:

Case: $\mathtt{t}$ is a constant

Immediate: $|Consts(\mathtt{t})| = |\{\mathtt{t}\}| = 1 = size(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{succ\ t_1}, \mathtt{pred\ t_1}, \text{or} \mathtt{iszero\ t_1}$

By the induction hypothesis, $|Consts(\mathtt{t_1})| \leq size(\mathtt{t_1})$. We now calculate as follows: $|Consts(\mathtt{t})| = |Consts(\mathtt{t_1})| \leq size(\mathtt{t_1}) < size(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{if\ t_1\ then\ t_2\ else\ t_3}$

By the induction hypothesis, $|Consts(\mathtt{t_1})| \leq size(\mathtt{t_1})$ and $|Consts(\mathtt{t_2})| \leq size(\mathtt{t_2})$ and $|Consts(\mathtt{t_3})| \leq size(\mathtt{t_3})$. We now calculate as follows: $|Consts(\mathtt{t})| = |Consts(\mathtt{t_1}) \cup Consts(\mathtt{t_2}) \cup Consts(\mathtt{t_3})| \leq |Consts(\mathtt{t_1})| + |Consts(\mathtt{t_2})| + |Consts(\mathtt{t_3})| \leq size(\mathtt{t_1}) + size(\mathtt{t_2}) + size(\mathtt{t_3}) < size(\mathtt{t})$. □

The form of this proof can be clarified by restating it as a general reasoning principle. (Compare this principle with the induction principle for natural numbers on p. 10.)

3.2.1.8 Theorem [Principle of induction on terms]: Suppose that P is some predicate on terms. If we can show, for each $\mathtt{s}$, that $(\forall \mathtt{r}.\ size(\mathtt{r}) < size(\mathtt{s}) \text{ implies } P(\mathtt{r}))$ implies $P(\mathtt{s})$, then we may conclude that $P(\mathtt{t})$ holds for every term $\mathtt{t}$. □

Proof: Exercise. □

3.2.2 Evaluation

3.2.2.1 Definition: The set of **values** is the subset of terms defined by the following abstract grammar:

`v ::=`	`true`	*value true*
	`false`	*value false*
	`0`	*zero value*
	`succ v`	*successor value*

□

3.2.2.2 Definition: The **one-step evaluation** relation $\longrightarrow$ is the smallest relation containing all instances of the following rules: *Evaluation:*

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \qquad \text{(EV-BETABOOLT)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \qquad \text{(EV-BETABOOLF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \qquad \text{(EV-IF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{succ } \texttt{t}_1 \longrightarrow \texttt{succ } \texttt{t}_1'} \qquad \text{(EV-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(EV-BETANATPZ)}$$

$$\texttt{pred (succ v)} \longrightarrow \texttt{v} \qquad \text{(EV-BETANATPS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{pred } \texttt{t}_1 \longrightarrow \texttt{pred } \texttt{t}_1'} \qquad \text{(EV-PRED)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(EV-BETANATIZ)}$$

$$\texttt{iszero (succ v)} \longrightarrow \texttt{false} \qquad \text{(EV-BETANATIS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{iszero } \texttt{t}_1 \longrightarrow \texttt{iszero } \texttt{t}_1'} \qquad \text{(EV-ISZERO)}$$

□

3.2.2.3 Definition: The **multi-step evaluation** relation $\longrightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that

- if $\texttt{t} \longrightarrow \texttt{t}'$ then $\texttt{t} \longrightarrow^* \texttt{t}'$,
- $\texttt{t} \longrightarrow^* \texttt{t}$ for all $\texttt{t}$, and
- if $\texttt{t} \longrightarrow^* \texttt{t}'$ and $\texttt{t}' \longrightarrow^* \texttt{t}''$, then $\texttt{t} \longrightarrow^* \texttt{t}''$. □

3.2.2.4 Definition: A term `t` is in **normal form** if no evaluation rule applies to it—i.e., if there is no $\texttt{t}'$ such that $\texttt{t} \longrightarrow \texttt{t}'$. □

3.2.2.5 Definition: An **evaluation sequence** starting from a term $\mathtt{t}$ is a (finite or infinite) sequence of terms $\mathtt{t}_1, \mathtt{t}_2, \ldots$, such that

$$\begin{array}{l} \mathtt{t} \longrightarrow \mathtt{t}_1 \\ \mathtt{t}_1 \longrightarrow \mathtt{t}_2 \\ \text{etc.} \end{array}$$ □

3.2.2.6 Definition: A term is said to be **stuck** if it is a normal form but not a value. □

3.2.2.7 Exercise: Write an abstract grammar that generates all (and only) the stuck arithmetic expressions. □

"Stuckness" gives us a simple notion of "run-time type error" for this rather abstract abstract machine. Intuitively, it characterizes the situations where the operational semantics does not know what to do because the program has reached a "meaningless state." A more serious implementation might choose other behavior in these cases, such as dumping core.

3.2.3 Properties

3.2.3.1 Proposition: Every value is in normal form. □

Proof: By inspection of the definitions of values and one-step evaluation. □

3.2.3.2 Proposition [Determinacy of evaluation]: If $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathtt{t} \longrightarrow \mathtt{t}''$, then $\mathtt{t}' = \mathtt{t}''$. □

Proof: By induction on the structure of $\mathtt{t}$. □

3.2.3.3 Definition: A value $\mathtt{v}$ is the **result** of a term $\mathtt{t}$ if $\mathtt{t} \longrightarrow^* \mathtt{v}$. □

3.2.3.4 Proposition [Unique results]: If $\mathtt{v}$ and $\mathtt{w}$ are both results of $\mathtt{t}$, then $\mathtt{v} = \mathtt{w}$. □

3.3 An ML Implementation

Optimized for readability and for correspondence with the mathematical definitions, not for efficient execution!

We'll ignore parsing and printing, the top-level read-eval-print loop, etc. Interested readers are encouraged to have a look at the ML code for the whole typechecker.

3.3.1 Syntax

```
type info

type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

The `info` components of this datatype are extracted, as necessary, by the printing functions.

3.3.2 Evaluation

```
exception No

let rec eval1 t =
  match t with
    TmIf(fi,t1,t2,t3) when not (isval t1) →
      TmIf(fi, eval1 t1, t2, t3)
  | TmIf(fi,TmTrue(_),t2,t3) →
      t2
  | TmIf(fi,TmFalse(_),t2,t3) →
      t3
  | TmSucc(fi,t1) when not (isval t1) →
      TmSucc(fi, eval1 t1)
  | TmPred(fi,t1) when not (isval t1) →
      TmPred(fi, eval1 t1)
  | TmPred(_,TmZero(_)) →
      TmZero(unknown)
  | TmPred(_,TmSucc(_,v1)) →
      v1
  | TmIsZero(fi,t1) when not (isval t1) →
      TmIsZero(fi, eval1 t1)
  | TmIsZero(_,TmZero(_)) →
      TmTrue(unknown)
  | TmIsZero(_,TmSucc(_,_)) →
      TmFalse(unknown)
  | _ → raise No

let rec eval t =
  try eval (eval1 t) with No → t
```

3.4 Summary

$\boxed{+\mathbb{B}}$ **: Arithmetic expressions and Booleans**

Syntax:

`t`	::=	`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*
`v`	::=	`true`	*value true*
		`false`	*value false*
		`0`	*zero value*
		`succ v`	*successor value*

Evaluation:

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \qquad \text{(EV-BETABOOLT)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \qquad \text{(EV-BETABOOLF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \qquad \text{(EV-IF)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{succ } \texttt{t}_1 \longrightarrow \texttt{succ } \texttt{t}_1'} \qquad \text{(EV-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(EV-BETANATPZ)}$$

$$\texttt{pred (succ v)} \longrightarrow \texttt{v} \qquad \text{(EV-BETANATPS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{pred } \texttt{t}_1 \longrightarrow \texttt{pred } \texttt{t}_1'} \qquad \text{(EV-PRED)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(EV-BETANATIZ)}$$

$$\texttt{iszero (succ v)} \longrightarrow \texttt{false} \qquad \text{(EV-BETANATIS)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{iszero } \texttt{t}_1 \longrightarrow \texttt{iszero } \texttt{t}_1'} \qquad \text{(EV-ISZERO)}$$

Chapter 4

The Lambda-Calculus

There may, indeed, be other applications of the system than its use as a logic.
—Alonzo Church, 1932

In the mid 1960s, Peter Landin observed that a complex programming language can often be understood very cleanly by formulating it as a tiny "core calculus" capturing its essential mechanisms, together with a collection of convenient "derived forms" whose behavior is understood by translating them into the core [Lan64, Lan65, Lan66] (also cf. [Ten81]). The core language used by Landin was the **lambda-calculus**, a formal system in which all computation is reduced to the basic operations of function definition and application. Since the 60s, the lambda-calculus has seen widespread use in the specification of programming language features, language design and implementation, and the study of type systems. Its importance arises from the fact that it can be viewed simultaneously as a simple programming language *in* which computations can be described and as a mathematical object *about* which rigorous statements can be proved.

The lambda-calculus is just one of a large number of core calculi that have been used for these purposes. For example, the **pi-calculus** of Robin Milner, Joachim Parrow, and David Walker [MPW92, Mil91] has become a popular core language for defining the semantics of message-based concurrent languages, while Martín Abadi and Luca Cardelli's **object calculus** [AC96] distills the core features of many object-oriented languages. The concepts and techniques we will develop for the lambda-calculus can, in most cases, be transferred quite directly to other calculi.

The lambda-calculus can be enriched in a variety of ways. First, it is often convenient to add special concrete syntax for features like tuples and records whose behavior can already be simulated in the core language. More interestingly, we may want to provide more complex features such as mutable reference cells or nonlocal exception handling, which can be modeled in the core language only via rather heavy translations. Such extensions make the calculus look more like a full-blown high-level programming language in its own right, and lead eventually to languages such as ML [GMW79, MTH90, WAL$^+$89, MTHM97] and Scheme [SJ75, KCR98]. As we shall see, extensions to the core language often involve extensions to the type system as well.

This chapter reviews the definition and some basic properties of the **pure** or **untyped lambda-calculus**.

4.1 Basics

Procedural abstraction is a key feature of most programming languages. Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation abstractly, in terms of one or more named parameters; we then instantiate this function as needed, providing values for the parameters in each case. For example, it is second nature for a programmer to take a long and repetitive expression like

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
```

and rewrite it as

```
factorial(5)+factorial(7)-factorial(3),
```

where:

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

For each nonnegative number `n`, instantiating the function `factorial` with the argument `n` yields a number, the factorial of `n`, as result. Writing "`λn. ...`" as a shorthand for "the function that, for each `n`, yields...," we can restate the definition of `factorial` as:

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

Then `factorial(0)` means "the function '`λn. if n=0 then 1 else ...`' applied to the argument `0`," that is, "the value that results when the bound variable `n` in the function body '`λn. if n=0 then 1 else ...`' is replaced by `0`," that is "`if 0=0 then 1 else ...`," that is, `1`.

In the 1930s, Alonzo Church invented a mathematical system called the **lambda-calculus** (or λ-calculus) that embodies this kind of function definition and application in a pure form [Chu36, Chu41]. In the lambda-calculus *everything* is a function: the arguments accepted by functions are themselves functions and the result returned by a function is another function.

4.1.1 Syntax

The syntax of the lambda-calculus comprises three kinds of terms. A variable `x` by itself is a lambda-term; the application of a lambda-term t_1 to another lambda-term t_2, written $t_1\ t_2$, is a lambda-term; and the abstraction of a variable `x` from a lambda-term t_1, written $\lambda x.\ t_1$, is a lambda-term. These forms are summarized in the following abstract grammar:

t	::=	x	*variable*
		λx.t	*abstraction*
		t t	*application*

The letters `s`, `t`, and `u` (with or without subscripts) are used throughout to stand for arbitrary lambda-terms, while `x`, `y`, and `z` are used to stand for arbitrary variable names. We call `s`, `t`, `u`, `x`, `y`, and `z` **metavariables**: they are "variables" in the sense that they stand for terms of the lambda-calculus, and "meta" in the sense that they are part of the **metalanguage** (i.e. English plus ordinary mathematical notations) in which an **object language**, the lambda-calculus, is being discussed. Since the set of short names is limited, we will also sometimes use `x`, `y`, etc. as object-language variables, but the context of the discussion will always make it clear which is which. For example, in a statement like "The term `(λx. λy. x (y x))` has the form `(λz. s)`, where `z` = `x` and `s` = `λy. x (y x)`," the names `z` and `s` are metavariables, whereas `x` and `y` are object-language variables. A complete summary of metavariable conventions appears in Appendix B.

The words "lambda-term" (or just "term") and "lambda-expression" are often used as synonyms.

Note that what we've defined is only the **abstract syntax** of lambda-terms. In a full-scale programming language, we would also need to address **concrete syntax** issues of lexing, parsing, operator precedence, etc. For present purposes, though, it is better to avoid such issues by concentrating on the abstract syntax, using parentheses informally as necessary to clarify the tree structure of examples. Application is taken to "associate to the left"—that is, `t s u` is considered the same as `(t s) u`. Also, the bodies of abstractions are assumed to extend as far to the right as possible, so that writing `λm. λn. m n m` is the same as writing `λx. (λy. (x y) x)`.

The variable `x` is said to be **bound** in the body `t` of the abstraction `λx.t`. Conversely, we say that a variable `x` is **free** in a term `s` if `x` appears at some position in `s` where it is not bound by an enclosing abstraction on `x`. For example, `x` is free in `x y` and `λy. x y`, but not in `λx.x` or `λz. λx. λy. x (y z)`.

A term with no free variables is said to be **closed**; closed terms are also sometimes called **combinators**. The simplest combinator is called the **identity function**:

```
id = λx.x;
```

4.1.2 Operational Semantics

In its pure form, the lambda-calculus has no built-in constants or operators—no numbers, arithmetic operations, records, loops, sequencing, I/O, etc. The sole means by which terms "compute" is the application of functions to arguments, which is captured in the following evaluation rule, traditionally called **beta-reduction**.

$$(\lambda \mathtt{x}.\mathtt{t}_1)\ \mathtt{v}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_1 \qquad \text{(EV-BETA)}$$

This rule says that an application of the form $(\lambda \mathtt{x}.\mathtt{t}_1)\ \mathtt{v}_2$ (where the argument has already been evaluated to a value), is evaluated by substituting the argument $\mathtt{v}_2$ for the bound variable x in the body $\mathtt{t}_1$. For example, (λx.x) y evaluates to y and (λx. x y) (u r) evaluates to u r y, while $\underline{(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x})\ \mathtt{z}}\ \mathtt{w}$ evaluates (by the underlined redex) to (λy.z) w, which further evaluates to z.

4.2 Programming in the Lambda-Calculus

The lambda-calculus is much more powerful than its tiny definition might suggest. For example, there is no built-in provision for multi-argument functions, but it is easy to achieve the same effect using **higher-order functions** that yield functions as results.

Suppose that s is a term involving two free variables x and y and we want to write a function f that, for each pair (p,q) of arguments, yields the result of substituting p for x and q for y in s. Instead of writing f = λ(x,y).s, as we might in a higher-level programming language, we write f = λx.λy.s. That is, f is a function that, given a value p for x, yields a function that, given a value q for y, yields the desired result. We then apply f to its arguments one at a time, writing f p q, which reduces to $(\lambda \mathtt{y}.\{\mathtt{p} \mapsto \mathtt{x}\}\mathtt{s})\ \mathtt{q}$ and then to $\{\mathtt{q} \mapsto \mathtt{y}\}\{\mathtt{p} \mapsto \mathtt{x}\}\mathtt{s}$. This transformation of multi-argument functions into higher-order functions is often called **Currying** after its popularizer, Haskell Curry. (It was actually invented by Schönfinkel, but the term "Schönfinkeling" has not caught on.)

Another common language feature that can easily be encoded in the lambda-calculus is boolean values and conditionals. Define the terms tru and fls as follows:

```
tru = λt. λf. t;
fls = λt. λf. f;
```

The only way to "interact" with combinators is by applying them to other terms. For example, we can use application to define a combinator test with the property that test b m n reduces to m when b = tru and reduces to q when b = fls.

```
test = λl. λm. λn. l m n;
```

The `test` combinator does not actually do much: `(test b m n)` just reduces to `(b m n)`. In effect, the boolean value `b` itself is the conditional: it takes two arguments and chooses the first (if it is `tru`) or the second (if it is `fls`). For example, the term `(test tru m n)` reduces as follows:

$$
\begin{array}{lll}
 & \mathtt{test\ tru\ m\ n} & \\
= & \underline{\mathtt{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}}\ \mathtt{m\ n} & \text{by definition}\\
\longrightarrow & \underline{\mathtt{(\lambda m.\ \lambda n.\ tru\ m\ n)\ m}}\ \mathtt{n} & \text{reducing the underlined redex}\\
\longrightarrow & \underline{\mathtt{(\lambda n.\ tru\ m\ n)\ n}} & \text{reducing the underlined redex}\\
\longrightarrow & \mathtt{tru\ m\ n} & \text{reducing the underlined redex}\\
= & \underline{\mathtt{(\lambda t.\lambda f.t)\ m}}\ \mathtt{n} & \text{by definition}\\
\longrightarrow & \underline{\mathtt{(\lambda f.\ m)\ n}} & \text{reducing the underlined redex}\\
\longrightarrow & \mathtt{m} & \text{reducing the underlined redex}
\end{array}
$$

We can also write boolean operators like logical conjunction as functions:

```
and = λb. λc. b c fls;
```

That is, `and` is a function that, given two boolean arguments `b` and `c`, returns `c` if `b` is `tru` or `fls` if `b` is `fls`; thus `and b c` yields `tru` if both `b` and `c` are `tru` and `fls` if either `b` or `c` is `fls`.

```
  and tru tru;
▶ (λt. λf. t)

  and tru fls;
▶ (λt. λf. f)

  and fls tru;
▶ (λt. λf. f)

  and fls fls;
▶ (λt. λf. f)
```

4.2.1 Exercise: Define logical `or` and `not` functions. □

Using booleans, we can encode pairs of values as lambda-terms. Define:

```
pair = λf.λs.λb. b f s;
fst = λp. p tru;
snd = λp. p fls;
```

That is, `pair m n` is a function that, when applied to a boolean `b`, applies `b` to `m` and `n`. By the definition of booleans, this application yields `m` if `b` is `tru` and `n` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean. To check that `fst (pair m n)` $\longrightarrow^*$ `m`, calculate as follows:

$$
\begin{array}{rll}
 & \mathtt{fst\ (pair\ m\ n)} & \\
= & \mathtt{fst\ (\underline{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ m}\ n)} & \text{by definition} \\
\longrightarrow & \mathtt{fst\ (\underline{(\lambda s.\ \lambda b.\ b\ m\ s)\ n})} & \text{reducing the underlined redex} \\
\longrightarrow & \mathtt{fst\ (\lambda b.\ b\ m\ n)} & \text{reducing the underlined redex} \\
= & \mathtt{\underline{(\lambda p.\ p\ tru)\ (\lambda b.\ b\ m\ n)}} & \text{by definition} \\
\longrightarrow & \mathtt{\underline{(\lambda b.\ b\ m\ n)\ tru}} & \text{reducing the underlined redex} \\
\longrightarrow & \mathtt{tru\ m\ n} & \text{reducing the underlined redex} \\
\longrightarrow^* & \mathtt{m} & \text{as before.}
\end{array}
$$

The encoding of numbers as lambda-terms is only slightly more intricate than what we have just seen. Define the **Church numerals** $\mathtt{c}_0$, $\mathtt{c}_1$, $\mathtt{c}_2$, etc., as follows:

$$
\begin{array}{l}
\mathtt{c_0\ =\ \lambda s.\ \lambda z.\ z;} \\
\mathtt{c_1\ =\ \lambda s.\ \lambda z.\ s\ z;} \\
\mathtt{c_2\ =\ \lambda s.\ \lambda z.\ s\ (s\ z);} \\
\mathtt{c_3\ =\ \lambda s.\ \lambda z.\ s\ (s\ (s\ z));} \\
\mathtt{c_4\ =\ \lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ z)));} \\
\mathtt{etc.}
\end{array}
$$

That is, each number n is represented by a combinator $\mathtt{c}_n$ that takes two arguments, `z` and `s` (for "zero" and "successor"), and applies `s`, n times, to `z`. As with booleans and pairs, this encoding makes numbers into active entities: the number n is represented by a function that does something n times—a kind of active unary numeral.

We can define some common arithmetic operations on Church numerals as follows:

$$
\begin{array}{l}
\mathtt{plus\ =\ \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z);} \\
\mathtt{times\ =\ \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0;}
\end{array}
$$

Here, `plus` is a combinator that takes two Church numerals, `m` and `n`, as arguments, and yields another Church numeral—i.e., a function that accepts arguments `z` and `s`, applies `s` iterated n times to `z` (by passing `s` and `z` as arguments to `n`), and then applies `s` iterated m more times to the result.

4.2.2 Exercise: Verify that $\mathtt{(plus\ c_2\ c_1)} \longleftrightarrow^* \mathtt{c_3}$. □

The definition of `times` uses another trick: since `plus` takes its arguments one at a time, applying it to just one argument `n` yields the function that adds `n` to whatever argument it is given. Passing this function as the second argument to `m` and $\mathtt{c}_0$ as the first argument means "apply the function that adds n to its argument, iterated m times, to zero," i.e., "add together m copies of n."

4.2.3 Exercise [Recommended]: Define a similar term for calculating the successor of a number. (Solution on page 141.) □

4.2.4 Exercise: Define a similar term for raising one number to the power of another. □

To test whether a Church numeral is zero, we must apply it to a pair of terms `zz` and `ss` such that applying `ss` to `zz` one or more times yields `fls`, while not applying it at all yields `tru`. Clearly, we should take `zz` to be just `tru`. For `ss`, we use a function that throws away its argument and always returns `fls`.

```
iszro = λm. m (λx. fls) tru;
```

Surprisingly, it is quite a bit more difficult to subtract using Church numerals. It can be done using the following rather tricky "predecessor function," which, given c_0 as argument, returns c_0 and, given c_{i+1}, returns c_i:

```
zz = pair c₀ c₀;
ss = λp. pair (snd p) (plus c₁ (snd p));
prd = λm. fst (m ss zz);
```

This definition works by using `m` as a function to apply m copies of the function `ss` to the starting value `zz`. Each copy of `ss` takes a pair of numerals `pair` c_i c_j as its argument and yields (`pair` c_j c_{j+1}) as its result. So applying `ss` m times to (`pair` c_0 c_0) yields (`pair` c_0 c_0) when $m = 0$ and (`pair` c_{m-1} c_m) when `m` is positive. In both cases, the predecessor of `m` is found in the first component.

4.2.5 Exercise:

1. Use `prd` to define a subtraction function.

2. How many steps of evaluation (as a function of n) are required to calculate the result of (`prd` c_n)? □

4.2.6 Exercise: Write a function `equal` that tests two numbers for equality and returns a boolean. For example,

```
  equal c₃ c₃;
▶ (λt. λf. t)

  equal c₃ c₂;
▶ (λt. λf. f)
```

(Solution on page 141.) □

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques. Of course, in most programming languages based on the lambda-calculus, such basic data types are added as primitive constants, rather than being encoded.

4.2.7 Exercise [Recommended]: Build an encoding of lists in the pure lambda-calculus by defining a constant `nil` (the empty list) and operations `cons` (for constructing a list from a new head element and an old list), `head` and `tail` (for extracting the parts of a list), and `isnull` (for testing whether a list is empty).

One straightforward way to do this is to generalize the encoding of numbers as follows. A list is represented by a function that takes two arguments, `hh` and `tt`. If the list is empty, this function simply returns `tt`. On the other hand, if the list has head `h` and tail `t`, the function calls `hh`, passing it `h` and `(t hh tt)` as parameters. (In other words, a list is represented by its own `fold` function.)

Can you think of any different ways of encoding lists? (Solution on page 141.) □

Recall that a term that cannot take a step under the evaluation relation is said to be in **normal form**. Interestingly, in the untyped lambda-calculus, not every term can be evaluated to a normal form. For example, the **divergent** combinator

```
omega = (λx. x x) (λx. x x);
```

can never be reduced to a value. It contains just one redex, and reducing this redex yields exactly `omega` again! Terms with no normal form are sometimes said to **diverge**.

The `omega` combinator has a useful generalization called the **fixed-point combinator** (or Y-combinator), which can be used to define recursive functions such as `factorial`.

```
fixpoint = λf. (λx. f (λy. (x x) y))
               (λx. f (λy. (x x) y));
```

The crucial property of `fixpoint` is that `fixpoint ff` has the same behavior as `ff (fixpoint ff)` for any term `ff`.

(Note that this is a *call-by-value* fixed point combinator, since we using a call-by-value reduction strategy. The simpler call-by-name fixed point combinator

$$\texttt{fixpoint}_n \texttt{ = λf. (λx. f (x x))}$$
$$\texttt{(λx. f (x x));}$$

is useless in a call-by-value setting, since an expression like $\texttt{fixpoint}_n$ `ff` always diverges, no matter what `ff` is.)

Now, suppose we want to write a recursive function definition of the form `ff` $=$ $\langle$*body containing* `ff`$\rangle$—i.e., we want to write a definition where the term on the right-hand side of the $=$ uses the very function that we are defining, as in the definition of `factorial` on page 23. 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 `fixpoint` by defining $\mathtt{gg} = \lambda\mathtt{f}.\langle\textit{body containing}\ \mathtt{f}\rangle$ and $\mathtt{ff} = \mathtt{fixpoint\ gg}$. For example, we can define the factorial function by

```
ff = λf. λn.
       test
         (iszro n) (λx. c₁) (λx. (times n (f (prd n)))) c₀;
factorial = fixpoint ff;
```

We then have:

```
  equal (factorial c₃) c₆;
▸ (λt. λf. t)
```

4.2.8 Exercise [Recommended]: Use the `fixpoint` combinator and the encoding of lists from Exercise **??** to write a function that sums lists of church numerals. (Solution on page 142.) □

4.3 Formalities

As we did in Chapter 3, we must now consider the syntax and operational semantics of the lambda-calculus in a little more detail. Most of the structure we need is closely analogous to what we saw there. However, the operation of substituting a term for a variable involves some surprising subtleties.

4.3.1 Syntax

As in Chapter 3, the abstract grammar defining terms should be read as shorthand for the union of an inductively defined family of sets of abstract syntax trees. To define it precisely, we begin with some set $\mathcal{V}$ of variable names.

4.3.1.1 Definition [Terms]: The set of terms is the smallest set $\mathcal{T}$ such that

1. $\mathtt{x} \in \mathcal{T}$ for every $\mathtt{x} \in \mathcal{V}$;
2. if $\mathtt{t}_1 \in \mathcal{T}$ and $\mathtt{x}$ is a variable name, then $\lambda\mathtt{x}.\mathtt{t}_1 \in \mathcal{T}$;
3. if $\mathtt{t}_1 \in \mathcal{T}$ and $\mathtt{t}_2 \in \mathcal{T}$, then $(\mathtt{t}_1\ \mathtt{t}_2) \in \mathcal{T}$. □

The size of a term $\mathtt{t}$ can be defined exactly as we did for arithmetic expressions in Definition 3.2.1.6. More interestingly, we can give a simple inductive definition of the set of variables appearing "free" in a lambda-term.

4.3.1.2 Definition: The set of **free variables** of a term $\mathtt{t}$, written $FV(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl} FV(\mathtt{x}) & = & \{\mathtt{x}\} \\ FV(\lambda\mathtt{x}.\mathtt{t}_1) & = & FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\ FV(\mathtt{t}_1\ \mathtt{t}_2) & = & FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \end{array}$$

□

Here is an inductive proof of a simple fact relating the definitions of size and free variables, analogous to Lemma 3.2.1.7 in the previous chapter.

4.3.1.3 Lemma: The number of distinct free variables in a term $\mathtt{t}$ is always smaller than the size of $\mathtt{t}$ (in symbols: $|FV(\mathtt{t})| \leq size(\mathtt{t})$). □

Proof: By induction on the size of $\mathtt{t}$. (That is, assuming the desired property for all terms smaller than $\mathtt{t}$, we must prove it for $\mathtt{t}$ itself; if we can do this, we may conclude that the property holds for all $\mathtt{t}$.) There are three cases to consider:

Case: $\mathtt{t} = \mathtt{x}$

Immediate: $|FV(\mathtt{t})| = |\{\mathtt{x}\}| = 1 = size(\mathtt{t})$.

Case: $\mathtt{t} = \lambda\mathtt{x}.\mathtt{t}_1$

By the induction hypothesis, $|FV(\mathtt{t}_1)| \leq size(\mathtt{t}_1)$. We now calculate as follows: $|FV(\mathtt{t})| = |FV(\mathtt{t}_1) \setminus \{\mathtt{x}\}| \leq |FV(\mathtt{t}_1)| \leq size(\mathtt{t}_1) < size(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$

By the induction hypothesis, $|FV(\mathtt{t}_1)| \leq size(\mathtt{t}_1)$ and $|FV(\mathtt{t}_2)| \leq size(\mathtt{t}_2)$. We now calculate as follows: $|FV(\mathtt{t})| = |FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)| \leq |FV(\mathtt{t}_1)| + |FV(\mathtt{t}_2)| \leq size(\mathtt{t}_1) + size(\mathtt{t}_2) < size(\mathtt{t})$. □

4.3.2 Substitution

Dealing carefully with the operation of substitution requires some work. We'll take it in two large steps. First (in this section), we'll discuss the basic issues, identify some traps for the unwary, and come up with a definition of substitution that is precise enough for most purposes. In Section **??** we'll go the rest of the way, defining a more refined "nameless" presentation of terms on which substitution is completely formal (and, in particular, suitable for implementation). The reason for presenting both is that the nameless formulation becomes too fiddly to use all the time. Having worked it out in detail, we will return to the more comfortable, slightly informal presentation developed in this section, regarding it as a convenient shorthand for the other.

It is instructive to arrive at the correct definition of substitution via a couple of wrong attempts. First off, let's try the most naive possible recursive definition:

$$
\begin{array}{lcll}
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{x} & = & \mathtt{s} & \\
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y}.\mathtt{t}_1) & = & \lambda\mathtt{y}.\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1 & \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t}_1\ \mathtt{t}_2) & = & \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_2 &
\end{array}
$$

This definition works fine for most examples. For instance, it gives

$$\{\mathtt{x} \mapsto (\lambda\mathtt{z}.\ \mathtt{z}\ \mathtt{w})\}(\lambda\mathtt{y}.\ \mathtt{x}) = (\lambda\mathtt{y}.\lambda\mathtt{z}.\ \mathtt{z}\ \mathtt{w}),$$

which is fine. However, if we are unlucky with our choice of bound variable names, the definition breaks down. For example:

$$\{\mathtt{x} \mapsto \mathtt{y}\}(\lambda\mathtt{x.x}) = \lambda\mathtt{x.y}$$

This conflicts with the basic intuition about functional abstractions that *the alphabetic names of bound variables do not matter*—the identity function is exactly the same whether we write it λ`x.x` or λ`y.y` or λ`franz.franz`. If these do not behave exactly the same under substitution, then they will not behave the same under reduction either, which seems wrong.

Clearly, the first mistake that we've made in the naive definition of substitution is that we have not distinguished between *free* occurrences of a variable `x` in a term `t` (which should get replaced during substitution) and *bound* ones, which should not. When we reach an abstraction binding the name `x` inside of `t`, the substitution operation should stop. This leads to the next attempt at a definition of substitution:

$$\begin{array}{lcll} \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{x} & = & \mathtt{s} & \\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\ \{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y.t}_1) & = & \left\{ \begin{array}{l} \lambda\mathtt{y.}\ \mathtt{t}_1 \\ \lambda\mathtt{y.}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1 \end{array} \right. & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \end{array} \\ \{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t}_1\ \mathtt{t}_2) & = & \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_2 & \end{array}$$

This is better, but it is still not quite right. For example, consider what happens when we substitute the term `z` for the variable `x` in the term λ`z.x`:

$$\{\mathtt{x} \mapsto \mathtt{z}\}(\lambda\mathtt{z.x}) = \lambda\mathtt{z.z}$$

This time, we have made essentially the opposite mistake as in the previous unfortunate example: we've turned the constant function λ`z.x` into the identity function! Again, this occurred only because we happened to choose `z` as the name of the bound variable in the constant function, so something is clearly still wrong.

This phenomenon of free variables in a term `s` becoming bound when `s` is naively substituted into a term `t` containing variable binders is called **variable capture**. To avoid it, we need to make sure that the bound variable names of `t` kept distinct from the free variable names of `s`. A substitution operation that does this correctly is called **capture-avoiding substitution**. This is almost always what is meant by the unqualified term "substitution." We can achieve this by adding another side condition to the second clause of the abstraction case:

$$\begin{array}{lcll} \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{x} & = & \mathtt{s} & \\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\ \{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y.t}_1) & = & \left\{ \begin{array}{l} \lambda\mathtt{y.}\ \mathtt{t}_1 \\ \lambda\mathtt{y.}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1 \end{array} \right. & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \end{array} \\ \{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t}_1\ \mathtt{t}_2) & = & \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_1\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}_2 & \end{array}$$

We are almost there: this definition of substitution does the right thing *when it does anything at all*. The problem now is that our last fix has changed substitution into a partial operation. For example, the new definition does not give any result at all for $\{\mathtt{x} \mapsto \mathtt{y\ z}\}(\lambda\mathtt{y.\ x\ y})$: the bound variable $\mathtt{y}$ of the term being substituted into is not equal to $\mathtt{x}$, but it does appear free in $(\mathtt{y\ z})$, so none of the clauses of the definition apply.

One common fix for this last problem in the type systems and lambda-calculus literature is to work with terms "up to renaming of bound variables" (or "up to alpha-conversion"):

4.3.2.1 Convention: Terms that differ only in the names of bound variables are interchangeable in all contexts. □

What this means in practice is that the name of any bound variable can be changed to another name (consistently making the same change in the body), at any point where this is convenient. For example, if we want to calculate $\{\mathtt{x} \mapsto \mathtt{y\ z}\}(\lambda\mathtt{y.\ x\ y})$, we first rewrite $(\lambda\mathtt{y.\ x\ y})$ as, say, $(\lambda\mathtt{w.\ x\ w})$. We then calculate $\{\mathtt{x} \mapsto \mathtt{y\ z}\}(\lambda\mathtt{w.\ x\ w})$, giving $(\lambda\mathtt{w.\ y\ z\ w})$. This renders the substitution operation "as good as total," since whenever we find ourselves about to apply it to a collection of arguments for which it is undefined, we can perform one or more steps of renaming as necessary, so that the side conditions are satisfied.

Indeed, having adopted this convention, we can formulate the definition of substitution a little more tersely. The first clause for abstractions can be dropped, since we can always assume (renaming if necessary) that the bound variable $\mathtt{y}$ is different from both $\mathtt{x}$ and the free variables of $\mathtt{s}$. This yields the final form of the definition.

4.3.2.2 Definition [Substitution]:

$$\begin{array}{lcll}
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{x} & = & \mathtt{s} & \\
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y.t_1}) & = & \lambda\mathtt{y.}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} & \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t_1\ t_2}) & = & \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_2} &
\end{array}$$
□

The convention about implicit renaming of bound variables is is easy to work with, and its slight informality does not usually lead to trouble in practice. We shall adopt it in most of what follows. However, it is also important to know how to formulate basic operations such as substitution with complete rigor, so that we have a solid foundation to fall back on in case there is ever any question. We will see one way to accomplish this in Section **??**.

4.3.2.3 Exercise: Without looking forward to Section **??**, can you think of any other ways in which the incomplete definition of substitution might be repaired to give a complete operation, without resorting to a convention about implicit renaming of bound variables? □

4.3.3 Operational Semantics

4.3.3.1 Definition: The set of values is the subset of terms defined by the following abstract grammar:

$$\begin{array}{lll} \mathtt{v} & ::= \mathtt{x} & \textit{variable} \\ & \quad \lambda\mathtt{x.t} & \textit{abstraction value} \end{array}$$

□

4.3.3.2 Definition: The one-step evaluation relation $\longrightarrow$ is the smallest relation containing all instances of the following rules:

$$(\lambda\mathtt{x.t_1})\ \mathtt{v_2} \longrightarrow \{\mathtt{x} \mapsto \mathtt{v_2}\}\mathtt{t_1} \qquad \text{(EV-BETA)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1}'}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \qquad \text{(EV-APP1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2}'}{\mathtt{v_1\ t_2} \longrightarrow \mathtt{v_1\ t_2'}} \qquad \text{(EV-APP2)}$$

□

4.3.4 Summary

The foregoing definitions may be summarized compactly as follows.

$\boxed{\lambda_u}$ **: Untyped lambda-calculus**

Syntax:

$$\begin{array}{lll} \mathtt{t} & ::= \mathtt{x} & \textit{variable} \\ & \quad \lambda\mathtt{x.t} & \textit{abstraction} \\ & \quad \mathtt{t\ t} & \textit{application} \end{array}$$

$$\begin{array}{lll} \mathtt{v} & ::= \mathtt{x} & \textit{variable} \\ & \quad \lambda\mathtt{x.t} & \textit{abstraction value} \end{array}$$

Evaluation:

$$(\lambda\mathtt{x.t_1})\ \mathtt{v_2} \longrightarrow \{\mathtt{x} \mapsto \mathtt{v_2}\}\mathtt{t_1} \qquad \text{(EV-BETA)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1}'}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \qquad \text{(EV-APP1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2}'}{\mathtt{v_1\ t_2} \longrightarrow \mathtt{v_1\ t_2'}} \qquad \text{(EV-APP2)}$$

4.4 Further Reading

Information on the untyped lambda-calculus can be found in many places. The first and best is Barendregt's encyclopedic monograph, *The Lambda Calculus* [Bar84]. Hindley and Seldin's book [HS86] gives a more accessible treatment of some of the basic material. Barendregt's article in the Handbook of Theoretical Computer Science [Bar90] is a compact survey. Material on lambda-calculus can also be found in many textbooks on functional programming languages (e.g. [AS85, FWH92, PJL92]) and programming language semantics (e.g.[Sch86, Gun92, Win93, Mit96]).

Part II

First-Order Systems

Chapter 5

Typed Arithmetic Expressions

5.1 Syntax

5.1.1 Definition: The set of **types** for arithmetic expressions contains just the symbols `Bool` and `Nat`:

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

The metavariables `S`, `T`, `U`, etc. range over types. □

5.2 The Typing Relation

The type system for arithmetic expressions is defined by a set of rules assigning types to terms. We write "$\vdash \texttt{t} : \texttt{T}$" to mean "term `t` has type `T`."

For example, the fact that the term `0` has the type `Nat` is captured by the following axiom:

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

Similarly, the term $\texttt{succ t}_1$ has type `Nat` as long as $\texttt{t}_1$ has type `Nat`.

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

Likewise, `pred` yields a `Nat` when its argument has type `Nat`

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

and `iszero` yields a `Bool` when its argument has type `Nat`:

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

The boolean constants both have type `Bool`

$$\vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

The only interesting typing rule is the one for `if` expressions:

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

The metavariable `T` is used to indicate that the result of the `if` is the type of the `then`- and `else`- branches, and that this may be any type (either `Nat` or `Bool` or, when we get to calculi with more interesting sets of types, any other type whatsoever).

5.3 Properties of Typing and Reduction

5.3.1 Typing Derivations

Formally, the typing relation can be thought of as a simple logic for proving **typing assertions** (sometimes called **typing judgements**) of the form $\vdash \texttt{t} : \texttt{T}$, pronounced "the term `t` has the type `T`." The provable statements of this logic are defined as follows:

5.3.1.1 Definition: The **typing relation** is the least two-place relation containing all instances of the axioms T-ZERO, T-TRUE, and T-FALSE and closed under all instances of the remaining rules. (In the literature on type systems, the symbol $\in$ is often used instead of $:$ for the typing relation.) □

5.3.1.2 Definition: A **typing derivation** is a tree of instances of the typing rules. Each statement $\vdash \texttt{t} : \texttt{T}$ in the typing relation is justified by a typing derivation with conclusion $\vdash \texttt{t} : \texttt{T}$. We sometimes write $\mathcal{D} :: \mathcal{J}$ to indicate that $\mathcal{D}$ is a derivation tree with conclusion $\mathcal{J}$. When $\mathcal{D} :: \mathcal{J}$ for some $\mathcal{D}$, we say that $\mathcal{J}$ is **derivable**. □

5.3.1.3 Theorem [Principle of induction on typing derivations]: Suppose that `P` is some predicate on typing statements. If we can show, for each typing derivation $\mathcal{D} :: \mathcal{J}$, that $(\texttt{P}(\mathcal{J}')$, for all $\mathcal{D}' :: \mathcal{J}'$ with $\mathit{size}(\mathcal{D}') < \mathit{size}(\mathcal{D}))$ implies $\texttt{P}(\mathcal{J})$, then we may conclude that $\texttt{P}(\mathcal{J})$ holds for every derivable typing statement. □

5.3.2 Typechecking

5.3.2.1 Lemma [Inversion of the typing relation]:

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

$$\begin{array}{l} \Gamma \vdash \texttt{t}_1 : \texttt{Bool} \\ \Gamma \vdash \texttt{t}_2 : \texttt{R} \\ \Gamma \vdash \texttt{t}_3 : \texttt{R}. \end{array}$$

The inversion lemma is sometimes called the **generation lemma** for the typing relation, since, given a valid typing statement, it shows how proofs of this statement might have been generated. □

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

5.3.2.2 Theorem [Uniqueness of Types]: Each term t has at most one type. That is, if t is typeable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules of Definition **??**. □

Proof: Exercise. □

5.3.3 Type Soundness

The most important property of this type system (or any other) is **soundness**: well-typed programs do not "go wrong." A simple way of formalizing this fact is the observation that well-typed programs can only reduce to well-typed programs—that is, a well-typed program can never reach an ill-typed state at run-time.

5.3.3.1 Theorem [Preservation of types during evaluation]: If $\vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow^* \texttt{t}'$, then $\vdash \texttt{t}' : \texttt{T}$. □

Proof: Exercise. □

The preservation theorem is often called "subject reduction" (or "subject evaluation") in the literature—the intuition being that a typing statement $\vdash$ `t : T` can be thought of as a sentence: "`t` has type `T`." The term `t` is the subject of this sentence, and the subject reduction property then says that the truth of the sentence is stable under reduction of the subject.

Recall (from Definition3.2.2.6) that a "stuck" term is one that is in normal form but not a value.

5.3.3.2 Theorem [Progress]: If $\vdash$ `t : T` and `t` is not a value, then there is some `t'` such that `t` $\longrightarrow$ `t'`. □

Proof: Exercise. □

The combination of the latter property with the subject reduction theorem guarantees that "well-typed terms never get stuck." This combination of properties captures what is usually thought of as "type safety" or "type soundness."

5.3.3.3 Exercise [Recommended]: Having seen the subject reduction property, we may wonder whether the opposite ("subject *expansion*") property also holds. Is it always the case that, if `t` $\longrightarrow^*$ `t'` and $\vdash$ `t' : T`, then $\vdash$ `t : T`? If so, prove it. If not, give a counterexample. (Solution on page 142.) □

5.4 Implementation

```
type ty =
    TyBool
  | TyNat

let rec tyeqv tyS tyT =
  match (tyS,tyT) with
    (TyBool,TyBool) → true
  | (TyNat,TyNat) → true
  | _ → false

val error : info → string → 'a

let rec typeof t =
  match t with
    TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
     TyBool
  | TmIf(fi,s1,s2,s3) →
     if tyeqv (typeof s1) TyBool then
       let tyS = typeof s2 in
```

```
    if tyeqv tyS (typeof s3) then tyS
    else error fi "arms of conditional have different types"
  else error fi "guard of conditional not a boolean"
| TmZero(fi) →
   TyNat
| TmSucc(fi,t1) →
   if tyeqv (typeof t1) TyNat then TyNat
   else error fi "argument of succ is not a number"
| TmPred(fi,t1) →
   if tyeqv (typeof t1) TyNat then TyNat
   else error fi "argument of pred is not a number"
| TmIsZero(fi,t1) →
   if tyeqv (typeof t1) TyNat then TyBool
   else error fi "argument of iszero is not a number"
```

Chapter 6

Simple Types

This chapter introduces the most elementary member of the family of typed lambda-calculi that we shall be studying for the rest of the course: the simply typed lambda-calculus of Church [Chu40] and Curry [CF58].

6.1 Syntax

6.1.1 Definition: The set of **simple types** over the atomic type `Bool` (for brevity, we omit natural numbers) is generated by the following grammar:

`T`	::=	`Bool`	*type of booleans*
		`T→T`	*type of functions*

The type constructor → is right-associative: `S→T→U` stands for `S→(T→U)`. □

6.1.2 Definition: The abstract syntax of **simply typed lambda-terms** (with booleans and conditional) is defined by the following grammar:

`t`	::=	`true`	*constant true*
		`false`	*constant false*
		`if t then t else t`	*conditional*
		`x`	*variable*
		`λx:T.t`	*abstraction*
		`t t`	*application*

□

In the literature on type systems, two different presentation styles have are commonly used:

- In **implicitly typed** (or, for historical reasons, **Curry-style**) systems, the pure (untyped) lambda-calculus is used as the term language. The typing rules define a relation between untyped terms and the types that classify them.
- In **explicitly typed** (or **Church-style**) systems, the term language itself is refined so that terms carry some type information within them; for example, the bound variables in function abstractions are always annotated with the type of the expected parameter. The type system relates typed terms and their types.

To a large degree, the choice is a matter of taste, though explicitly typed systems generally pose fewer algorithmic problems for typecheckers. We will adopt an explicitly typed presentation throughout.

6.2 The Typing Relation

In order to assign a type to an abstraction like $\lambda\mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_1$, we need to know what will happen *later* when it is applied to some argument $\mathtt{t}_2$. The annotation on the bound variable tells us that we may assume that the argument will be of type $\mathtt{T}_2$. In other words, the type of the result will be just the type of $\mathtt{t}_1$, where occurrences of x in $\mathtt{t}_1$ are assumed to denote terms of type $\mathtt{T}_2$. This intuition is captured by the following rule:

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

Since, in general, function abstractions can be nested, typing assertions actually have the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, pronounced "term t has type T under the assumptions Γ about the types of its free variables." Formally, the **typing context** Γ is just a list of variables and their types, and the "comma" operator extends Γ by concatenating a new binding on the right. To avoid confusion between the new binding and any bindings that may already appear in Γ, we require that the name x be chosen so that it does not already appear in $dom(\Gamma)$. (As usual, this condition can always be satisfied by renaming the bound variable if necessary.) Γ can thus be thought of as a finite function from variables to their types. Following this intuition, we will write $dom(\Gamma)$ for the set of variables bound by Γ and $\Gamma(\mathtt{x})$ for the type T associated with x in Γ.

So the rule for typing abstractions actually has the general form

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

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

The typing rule for variables follows immediately from this discussion. A variable has whatever type we are currently assuming it to have:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \tag{T-VAR}$$

Next, we need a rule for application:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 {\to} \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \tag{T-APP}$$

In English: If $\mathtt{t}_1$ evaluates to a function mapping arguments in $\mathtt{T}_2$ to results in $\mathtt{T}_1$ (under the assumption that the terms represented by its free variables yield results of the types associated to them by Γ), and if $\mathtt{t}_2$ evaluates to a result in $\mathtt{T}_2$, then the result of applying $\mathtt{t}_1$ to $\mathtt{t}_2$ will be a value of type $\mathtt{T}_1$.

We have now given typing rules for each of the individual constructs in our simple language. To assign types to whole programs, we combine these rules into derivation trees. For example, here is a derivation tree showing that the term `(λx:Bool.x) true` has type `Bool` in the empty context:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x{:}Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ \text{T-VAR}}{\vdash \lambda\mathtt{x{:}Bool.x} : \mathtt{Bool}{\to}\mathtt{Bool}}\ \text{T-ABS} \qquad \dfrac{}{\vdash \mathtt{true} : \mathtt{Bool}}\ \text{T-TRUE}}{\vdash (\lambda\mathtt{x{:}Bool.x})\ \mathtt{true} : \mathtt{Bool}}\ \text{T-APP}$$

6.2.1 Exercise [Quick check]: Show (by exhibiting derivation trees) that the following terms have the indicated types in the given contexts:

1. `f:Bool→Bool` ⊢ `f (if false then true else false) : Bool`
2. `f:Bool→Bool` ⊢ `λx:Bool. f (if x then false else x) : Bool→Bool` □

6.2.2 Exercise [Quick check]: Find a context Γ under which the term `f x y` has type `Bool`. Can you give a simple description of the set of *all* such contexts? □

It is interesting to note that, in the above discussion, the "→" type constructor comes with typing rules of two kinds:

1. an **introduction rule** describing how elements of the type can be created, and
2. an **elimination rule** describing how elements of the type can be used.

This terminology, which arises from connections between type theory and logic, is frequently useful in discussing type systems. When we come to consider more complex systems later in the course, we'll see a similar pattern of linked introduction and elimination rules being repeated for each type constructor we consider.

6.2.3 Exercise [Quick check]: Which of the rules for the type `Bool` are introduction rules and which are elimination rules? (Solution on page 142.) □

6.3 Summary

We have been discussing the simply typed lambda-calculus with booleans and conditionals, $\lambda{\to}\mathbb{B}$. In later chapters, we are going to want to extend the simply typed lambda-calculus with many other base types, in addition to or instead of booleans. We therefore split the formal summary of the system into two pieces: the **pure** simply typed lambda calculus $\lambda{\to}$, with no base types at all, and a separate extension with booleans.

$\boxed{\lambda{\to}}$ **: Simply typed lambda-calculus** **Extends** λ_u

Syntax:

$\mathtt{t}$	$::=$	$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}\mathtt{:T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{v}$	$::=$	$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}\mathtt{:T}.\mathtt{t}$	*abstraction value*
$\mathtt{T}$	$::=$	$\mathtt{T}{\to}\mathtt{T}$	*type of functions*
Γ	$::=$	$\emptyset$	*empty context*
		$\Gamma, \mathtt{x}\mathtt{:T}$	*term variable binding*

Evaluation:

$$(\lambda\mathtt{x}\mathtt{:T_2}.\mathtt{t_1})\ \mathtt{v_2} \longrightarrow \{\mathtt{x} \mapsto \mathtt{v_2}\}\mathtt{t_1} \qquad \text{(EV-BETA)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1}'}{\mathtt{t_1}\ \mathtt{t_2} \longrightarrow \mathtt{t_1'}\ \mathtt{t_2}} \qquad \text{(EV-APP1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2}'}{\mathtt{v_1}\ \mathtt{t_2} \longrightarrow \mathtt{v_1}\ \mathtt{t_2'}} \qquad \text{(EV-APP2)}$$

Typing:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}\mathtt{:T_2} \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \vdash \lambda\mathtt{x}\mathtt{:T_2}.\mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_1}} \qquad \text{(T-APP)}$$

The highlighted areas here are used to mark material that is new with respect to the untyped lambda-calculus—whole new rules as well as new bits that need to be added to rules that we have already seen.

6.3.1 Exercise [Quick check]: The system $\lambda_{\rightarrow}$ by itself is actually trivial, in the sense that it has no well-typed programs at all. Why? (Solution on page 142.) □

$\boxed{\lambda_{\rightarrow}\mathbb{B}}$ **: Simply typed lambda-calculus with booleans** **Extends $\lambda_{\rightarrow}$**

New syntactic forms:

`t`	::=	...	
		`true`	*constant true*
		`false`	*constant false*
		`if t then t else t`	*conditional*
`v`	::=	...	
		`true`	*value true*
		`false`	*value false*
`T`	::=	...	
		`Bool`	*type of booleans*

New evaluation rules:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(EV-BETABOOLT)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(EV-BETABOOLF)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(EV-IF)}$$

New typing rules:

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-IF)}$$

6.4 Properties of Typing and Reduction

6.4.1 Typechecking

As in Chapter 5, we need to develop a few basic lemmas before we can prove type soundness. Most of these are similar to what we saw before (just adding contexts to the typing relation and adding appropriate clauses for λ-terms. The only significant new requirement is a substitution principle for the typing relation (Lemma 6.4.2.1).

6.4.1.1 Lemma [Inversion of the typing relation]:

1. If $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\Gamma(\mathtt{x}) = \mathtt{R}$.
2. If $\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_2.\ \mathtt{t}_1 : \mathtt{R}_2 \to \mathtt{R}_1$, then $\mathtt{R}_2 = \mathtt{T}_2$ and $\Gamma, \mathtt{x}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{R}_1$.
3. If $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{R}$, then there is some type $\mathtt{T}_2$ such that $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{R}$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$.
4. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
5. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
6. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then

$$\begin{array}{l} \Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \\ \Gamma \vdash \mathtt{t}_2 : \mathtt{R} \\ \Gamma \vdash \mathtt{t}_3 : \mathtt{R}. \end{array}$$ □

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

6.4.1.2 Exercise [Easy]: Write out a typechecking algorithm for λ→ in pseudo-code or a programming language of your choice. Compare your answer with the ML implementation in Section 6.5. □

In Section 6.1, we chose to use an explicitly typed presentation of the calculus, partly in order to simplify the algorithmic issues involved in 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, the substance of which is that well-typed terms are in one-to-one correspondence with the typing derivations that justify their well-typedness (in a given environment). The typing derivation can be recovered immediately from the term, and vice versa. In fact, the correspondence is so straightforward that, in a sense, there is little difference between the term and the derivation. (For many of the type systems that we will see, this simple correspondence will not hold: there will be significant work involved in showing that typing derivations can be recovered effectively from typed terms.)

6.4.1.3 Theorem [Uniqueness of Types]: 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 typeable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules of Definition **??**. □

The proof of the uniqueness theorem is so direct that there is almost nothing to say. We present a few cases carefully just to illustrate the structure of proofs by induction on typing derivations.

Proof: Suppose that $\Gamma \vdash \mathtt{t} : \mathtt{S}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}$. We show, by induction on a derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$, that $\mathtt{S} = \mathtt{T}$.

Case T-VAR*:* $\mathtt{t} = \mathtt{x}$
with $\Gamma(\mathtt{x}) = \mathtt{T}$

By case (1) of the inversion lemma (6.4.1.1), the final rule in any derivation of $\Gamma \vdash \mathtt{t} : \mathtt{S}$ must also be T-VAR, and $\mathtt{S} = \mathtt{T}$.

Case T-ABS*:* $\mathtt{t} = \lambda \mathtt{y}{:}\mathtt{T_2}.\mathtt{t_1}$
$\mathtt{T} = \mathtt{T_2} \to \mathtt{T_1}$
$\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{T_1}$

By case (2) of the inversion lemma, the final rule in any derivation of $\Gamma \vdash \mathtt{t} : \mathtt{S}$ must also be T-ABS, and this derivation must have a subderivation with conclusion $\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{S_1}$, with $\mathtt{S} = \mathtt{T_2} \to \mathtt{S_1}$. By the induction hypothesis (on the subderivation with conclusion $(\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{T_1})$, we obtain $\mathtt{S_1} = \mathtt{T_1}$, from which $\mathtt{S} = \mathtt{T}$ is immediate.

Case T-APP, T-TRUE, T-FALSE, T-IF*:*

Similar. □

6.4.1.4 Exercise [Quick check]: Write out the case for T-APP. □

6.4.1.5 Exercise [Recommended]: Is there any context Γ and type $\mathtt{T}$ such that $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathtt{T}$? If so, give Γ and $\mathtt{T}$ and show a typing derivation for $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathtt{T}$; if not, prove it. □

We shall sometimes need to talk about the types of subterms of well-typed terms. The following notation is useful for this purpose.

6.4.1.6 Definition: Suppose the term $\mathtt{s}$ is well typed with respect to the typing context Γ, and that $\mathtt{r}$ is some subterm of $\mathtt{s}$. We write $\mathit{typeOf}_\Gamma(\mathtt{r})$ for the type $\mathtt{R}$ such that $\Gamma' \vdash \mathtt{r} : \mathtt{R}$, where Γ' is calculated by extending Γ with bindings corresponding to all of the function abstractions inside $\mathtt{s}$ under which $\mathtt{r}$ occurs. For example, if $\mathtt{s} = \lambda \mathtt{f}{:}\mathtt{A} \to \mathtt{B}.\ \lambda \mathtt{g}{:}\mathtt{A}.\ \mathtt{f}\ \mathtt{g}$ and $\mathtt{r} = \mathtt{f}\ \mathtt{g}$, then $\mathit{typeOf}_\Gamma(\mathtt{r}) = \mathtt{B}$. When (as in this chapter) Γ is fixed for the whole discussion, we usually elide it, writing just $\mathit{typeOf}(\mathtt{r})$.

Note that $\mathtt{R}$ is unique, by Theorem 6.4.1.3. □

We close this section with 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.

6.4.1.7 Lemma [Permutation]: If $\Gamma \vdash \texttt{t : T}$ and Δ is a permutation of Γ, then $\Delta \vdash \texttt{t : T}$. Moreover, the latter derivation has the same depth as the former. □

Proof: Straightforward induction on typing derivations. □

6.4.1.8 Lemma [Weakening]: If $\Gamma \vdash \texttt{t : T}$ and $\texttt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \texttt{x:S} \vdash \texttt{t : T}$. Moreover, the latter derivation has the same depth as the former. □

Proof: Straightforward induction on typing derivations. □

6.4.2 Typing and Substitution

6.4.2.1 Lemma [Substitution]: If $\Gamma, \texttt{x:S} \vdash \texttt{t : T}$ and $\Gamma \vdash \texttt{s : S}$, then $\Gamma \vdash \{\texttt{x} \mapsto \texttt{s}\}\texttt{t : T}$. □

6.4.2.2 Exercise [Recommended]: Prove the substitution lemma, using an induction on the depth of typing derivations and Lemma 6.4.1.1. The full proof appears in the solutions; try to write it out yourself before having a look at the answer. (Solution on page 142.) □

6.4.3 Type Soundness

6.4.3.1 Theorem [Preservation of types during evaluation]: If $\Gamma \vdash \texttt{t : T}$ and $\texttt{t} \longrightarrow^* \texttt{t}'$, then $\vdash \texttt{t}' \texttt{ : T}$. □

Proof: Exercise. □

6.4.3.2 Theorem [Progress]: If $\Gamma \vdash \texttt{t : T}$ and t is not a value, then there is some t′ such that $\texttt{t} \longrightarrow \texttt{t}'$. □

Proof: Exercise. □

6.5 Implementation

```
type ty =
    TyBool
  | TyArr of ty * ty

type term =
    TmTrue of info
  | TmFalse of info
```

```
  | TmIf of info * term * term * term
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term

let rec tyeqv tyS tyT =
  match (tyS,tyT) with
    (TyBool,TyBool) → true
  | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
       (tyeqv tyS1 tyT1) && (tyeqv tyS2 tyT2)
  | _ → false

let rec typeof ctx t =
  match t with
    TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
     TyBool
  | TmIf(fi,s1,s2,s3) →
     if tyeqv (typeof ctx s1) TyBool then
       let tyS = typeof ctx s2 in
       if tyeqv tyS (typeof ctx s3) then tyS
       else error fi "arms of conditional have different types"
     else error fi "guard of conditional not a boolean"
  | TmVar(fi,i,_) →
      gettype fi ctx i
  | TmAbs(fi,x,tyS,t1) →
      let ctx' = addbinding ctx x (VarBind(tyS)) in
      let tyT = typeof ctx' t1 in
      TyArr(tyS, tyshift tyT (-1))
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if tyeqv tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
```

6.6 Further Reading

The untyped lambda-calculus was developed by Church and his co-workers in the 1930s [Chu41]. The standard text for all aspects of the untyped lambda-calculus is Barendregt [Bar84]. Hindley and Seldin [HS86] is less comprehensive, but somewhat more accessible. Barendregt's article in the Handbook of Theoretical Computer Science [Bar90] is a compact survey.

The simply typed lambda-calculus is studied in Hindley and Seldin [HS86] and in even greater detail in Hindley's more recent book [**?**]. Material on both typed and untyped lambda-calculus can also be found in many textbooks on functional programming languages (e.g. [PJL92]) and programming language semantics (e.g.[Sch86, Gun92, Win93, Mit96]).

Chapter 7

Extensions

7.1 Base Types

□: **Extends**

New syntactic forms:

T	::=	...	
		X	*atomic type*

7.2 Unit type

□: **Extends**

New syntactic forms:

t	::=	...	
		unit	*constant* unit
T	::=	...	
		Unit	*unit type*

New typing rules:

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

Note: C and Java's `void` type is a misnomer – what they really mean is `Unit`

7.3 Let bindings

□: **Extends**

New syntactic forms:

$$\begin{array}{lll} \texttt{t} & ::= & \ldots \\ & & \texttt{let x=t in t} \end{array} \qquad \textit{let binding}$$

New typing rules:

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

7.3.1 Exercise [Recommended]: Add `let` bindings to the "`simple`" typechecker implementation. (Solution on page **??**.) □

7.4 Records and Tuples

□: **Extends**

New syntactic forms:

$$\begin{array}{lll} \texttt{t} & ::= & \ldots \\ & & \{\texttt{l}_i\texttt{=t}_i\ ^{i\in 1..n}\} \qquad \textit{record} \\ & & \texttt{t.l} \qquad \textit{projection} \end{array}$$

$$\begin{array}{lll} \texttt{v} & ::= & \ldots \\ & & \{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..n}\} \qquad \textit{record value} \end{array}$$

$$\begin{array}{lll} \texttt{T} & ::= & \ldots \\ & & \{\texttt{l}_i\texttt{:T}_i\ ^{i\in 1..n}\} \qquad \textit{type of records} \end{array}$$

New evaluation rules:

$$\{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..n}\}\texttt{.l}_i \longrightarrow \texttt{v}_i \qquad \text{(EV-RCDPROJ)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1\texttt{.l} \longrightarrow \texttt{t}_1'\texttt{.l}} \qquad \text{(EV-PROJ)}$$

$$\frac{\texttt{t}_j \longrightarrow \texttt{t}_j'}{\{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..n}\} \longrightarrow \{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..j-1}\texttt{,l}_j\texttt{=t}_j'\texttt{,l}_k\texttt{=t}_k\ ^{k\in j+1..n}\}} \qquad \text{(EV-RECORD)}$$

New typing rules:

$$\frac{\Gamma \vdash {}^{i \in 1..n}\ \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=}\mathtt{t}_i{}^{i \in 1..n}\} : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i{}^{i \in 1..n}\}} \qquad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i{}^{i \in 1..n}\}}{\Gamma \vdash \mathtt{t.l}_j : \mathtt{T}_j} \qquad \text{(T-PROJ)}$$

New abbreviations:

$$\{\ldots\mathtt{t}_i\ldots\} \stackrel{\text{def}}{=} \{\ldots\mathtt{i=t}_i\ldots\}$$

$$\{\ldots\mathtt{T}_i\ldots\} \stackrel{\text{def}}{=} \{\ldots\mathtt{i:T}_i\ldots\}$$

7.4.1 Exercise: In this presentation of records, the projection operation is used to extract the fields of a record one by one. Many high-level programming languages provide an alternative **pattern matching** syntax that extracts all the fields at the same time, allowing many programs to be expressed more concisely. Patterns can typically be nested, allowing parts to be extracted easily from complex nested data structures.

For example, we can add a simple form of pattern matching to the untyped lambda calculus with records by adding a new syntactic category of patterns, plus one new case (for the pattern matching construct itself) to the syntax of terms. To define the computation rule for pattern matching, we need an auxiliary "matching" function. Given a pattern and a term, the pattern matching function either fails (if the term does not match the pattern) or else yields a substitution that replaces variables appearing in the pattern with the corresponding subterms of the given term. Here is the formal definition of the calculus, highlighting differences from the pure untyped lambda-calculus with records and ordinary let-binding:

□**:** **Extends**

New syntactic forms:

$\mathtt{p}$	::=	...	
		$\mathtt{x}$	*variable pattern*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{p}_i{}^{i \in 1..n}\}$	*record pattern*
$\mathtt{t}$	::=	...	
		$\mathtt{let\ p\text{=}t\ in\ t}$	*pattern binding*

Matching rules::

$$match(\mathtt{x}, \mathtt{t}) = \{\mathtt{x} \mapsto \mathtt{t}\} \qquad \text{(M-VAR)}$$

$$\frac{match(\mathtt{p}_i, \mathtt{t}_i) = \sigma_i}{match(\{\mathtt{l}_i\mathtt{=}\mathtt{p}_i{}^{i \in 1..n}\}, \{\mathtt{l}_i\mathtt{=}\mathtt{t}_i{}^{i \in 1..n}\}) = \sigma_1 \circ \cdots \circ \sigma_n} \qquad \text{(M-RCD)}$$

New abbreviations:

$$\{\ldots p_i \ldots\} \stackrel{\text{def}}{=} \{\ldots i{=}p_i \ldots\}$$

Your job is to add types to this calculus, in the style of the simply typed lambda-calculus:

1. Give typing rules for the new constructs (making any changes to the syntax you feel are necessary in the process).
2. Sketch a proof of subject reduction for the whole calculus by stating the sequence of major lemmas needed to carry out the proof. (You do not need to show proofs, just statements of the required lemmas in the correct order.)

(Solution on page **??**.) □

□**:** **Extends**

New syntactic forms:

p	::=	...	
		x	*variable pattern*
		$\{l_i{=}p_i\ ^{i\in 1..n}\}$	*record pattern*
t	::=	...	
		$\texttt{let}\ p{=}t\ \texttt{in}\ t$	*pattern binding*

New typing rules:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \vdash p : T_1 \Rightarrow \Delta \qquad \Gamma, \Delta \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ p{=}t_1\ \texttt{in}\ t_2 : T_2} \quad \text{(T-LET)}$$

Matching rules::

$$match(x, t) = \{x \mapsto t\} \quad \text{(M-VAR)}$$

$$\frac{match(p_i, t_i) = \sigma_i}{match(\{l_i{=}p_i\ ^{i\in 1..n}\}, \{l_i{=}t_i\ ^{i\in 1..n}\}) = \sigma_1 \circ \cdots \circ \sigma_n} \quad \text{(M-RCD)}$$

New abbreviations:

$$\{\ldots p_i \ldots\} \stackrel{\text{def}}{=} \{\ldots i{=}p_i \ldots\}$$

Pattern typing rules:

$$\vdash \texttt{x} : \texttt{T} \Rightarrow \texttt{x:T} \qquad \text{(P-VAR)}$$

$$\frac{\vdash \texttt{p}_i : \texttt{T}_i \Rightarrow \Delta_i \quad \text{for each } i}{\vdash \{\texttt{l}_i\texttt{=p}_i{}^{i\in 1..n}\} : \{\texttt{k}_j\texttt{:T}_j{}^{j\in 1..m}\} \Rightarrow \Delta_1, \ldots, \Delta_n} \qquad \text{(P-RCD)}$$

7.5 Variants

□ : **Extends**

New syntactic forms:

`t`	::=	...	
		`<l=t> as T`	*tagging*
		`case t of ...|`$\texttt{l}_i$`=x⇒`$\texttt{t}_i$`|...`	*case*
`T`	::=	...	
		$\texttt{<l}_i\texttt{:T}_i{}^{i\in 1..n}\texttt{>}$	*type of variants*

7.6 General recursion

7.7 Lists

7.8 Lazy records and let-bindings

Chapter 8

Computational Effects

8.1 References

8.1.1 Summary

□ : **Extends**

Syntax:

t	::=	$\{l_i = t_i\ ^{i \in 1..n}\}$	*record*
		t.l	*projection*
		t as T	*coercion*
		x	*variable*
		fix t	*fixed point of t*
		inert[T]	*inert element of type T*
		let x=t in t	*let binding*
		true	*constant true*
		false	*constant false*
		if t then t else t	*conditional*
		λx:T.t	*abstraction*
		t t	*application*
		unit	*constant* unit
		l	*store location*
		ref t	*reference creation*
		!t	*dereference*
		t:=t	*assignment*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*

$$
\begin{array}{lcll}
\mathtt{v} & ::= & \{\mathtt{l}_i\mathtt{=v}_i \;^{i \in 1..n}\} & \textit{record value} \\
 & & \mathtt{x} & \textit{variable} \\
 & & \mathtt{true} & \textit{value true} \\
 & & \mathtt{false} & \textit{value false} \\
 & & \lambda\mathtt{x:T.t} & \textit{abstraction value} \\
 & & \mathtt{l} & \textit{store location} \\
 & & \mathtt{0} & \textit{zero value} \\
 & & \mathtt{succ\ v} & \textit{successor value} \\
\mathtt{T} & ::= & \{\mathtt{l}_i\mathtt{:T}_i \;^{i \in 1..n}\} & \textit{type of records} \\
 & & \mathtt{Bool} & \textit{type of booleans} \\
 & & \mathtt{Unit} & \textit{unit type} \\
 & & \mathtt{Ref\ T_1} & \textit{type of reference cells} \\
 & & \mathtt{X} & \textit{type variable} \\
 & & \mathtt{Nat} & \textit{type of natural numbers} \\
 & & \mathtt{T}\rightarrow\mathtt{T} & \textit{type of functions} \\
\Gamma & ::= & \emptyset & \textit{empty context} \\
 & & \Gamma, \mathtt{x:T} & \textit{term variable binding} \\
 & & \Gamma, \mathtt{X} & \textit{type variable binding} \\
\mu & ::= & \emptyset & \textit{empty store} \\
 & & \mu, \mathtt{l} = \mathtt{t} & \textit{location binding} \\
\Sigma & ::= & \emptyset & \textit{empty store typing} \\
 & & \Sigma, \mathtt{l:T} & \textit{location typing}
\end{array}
$$

Evaluation:

$$\{\mathtt{l}_i\mathtt{=v}_i \;^{i \in 1..n}\}\mathtt{.l}_i \mid \mu \longrightarrow \mathtt{v}_i \mid \mu \qquad \text{(EV-RCDPROJ)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\mathtt{.l} \mid \mu \longrightarrow \mathtt{t}_1'\mathtt{.l} \mid \mu'} \qquad \text{(EV-PROJ)}$$

$$\frac{\mathtt{t}_j \mid \mu \longrightarrow \mathtt{t}_j' \mid \mu'}{\{\mathtt{l}_i\mathtt{=v}_i \;^{i \in 1..n}\} \mid \mu \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i \;^{i \in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j'\mathtt{,l}_k\mathtt{=t}_k \;^{k \in j+1..n}\} \mid \mu'} \qquad \text{(EV-RECORD)}$$

$$\mathtt{t}_1 \ \mathtt{as}\ \mathtt{T}_2 \mid \mu \longrightarrow \mathtt{t}_1 \mid \mu \qquad \text{(EV-COERCE1)}$$

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \mid \mu \longrightarrow \mathtt{t}_2 \mid \mu \qquad \text{(EV-BETABOOLT)}$$

$$\mathtt{if\ false\ then\ t_2\ else\ t_3} \mid \mu \longrightarrow \mathtt{t}_3 \mid \mu \qquad \text{(EV-BETABOOLF)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \mid \mu \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3} \mid \mu'} \qquad \text{(EV-IF)}$$

$$(\lambda\mathtt{x:T_2.t_1})\ \mathtt{v}_2 \mid \mu \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_1 \mid \mu \qquad \text{(EV-BETA)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1{}' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \qquad \text{(EV-APP1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2{}' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \qquad \text{(Ev-App2)}$$

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l = v_1)} \qquad \text{(Ev-Ref)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1{}' \mid \mu'}{\texttt{ref}\ t_1 \mid \mu \longrightarrow \texttt{ref}\ t_1' \mid \mu'} \qquad \text{(Ev-Ref1)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \qquad \text{(Ev-Deref)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1{}' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(Ev-Deref1)}$$

$$l{:=}v_2 \mid \mu \longrightarrow \texttt{unit} \mid \{l \mapsto v_2\}\mu \qquad \text{(Ev-Assign)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1{}' \mid \mu'}{t_1{:=}t_2 \mid \mu \longrightarrow t_1'{:=}t_2 \mid \mu'} \qquad \text{(Ev-Assign1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2{}' \mid \mu'}{v_1{:=}t_2 \mid \mu \longrightarrow v_1{:=}t_2' \mid \mu'} \qquad \text{(Ev-Assign2)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{succ}\ t_1 \mid \mu \longrightarrow \texttt{succ}\ t_1' \mid \mu'} \qquad \text{(Ev-Succ)}$$

$$\texttt{pred}\ 0 \mid \mu \longrightarrow 0 \mid \mu \qquad \text{(Ev-BetaNatPZ)}$$

$$\texttt{pred}\ (\texttt{succ}\ v) \mid \mu \longrightarrow v \mid \mu \qquad \text{(Ev-BetaNatPS)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{pred}\ t_1 \mid \mu \longrightarrow \texttt{pred}\ t_1' \mid \mu} \qquad \text{(Ev-Pred)}$$

$$\texttt{iszero}\ 0 \mid \mu \longrightarrow \texttt{true} \mid \mu \qquad \text{(Ev-BetaNatIZ)}$$

$$\texttt{iszero}\ (\texttt{succ}\ v) \mid \mu \longrightarrow \texttt{false} \mid \mu \qquad \text{(Ev-BetaNatIS)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{iszero}\ t_1 \mid \mu \longrightarrow \texttt{iszero}\ t_1' \mid \mu'} \qquad \text{(Ev-IsZero)}$$

Typing:

$$\frac{\Gamma \vdash {}^{i \in 1..n}\ \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ {}^{i \in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ {}^{i \in 1..n}\}} \qquad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\mathtt{l}_i\mathtt{:T}_i\ {}^{i \in 1..n}\}}{\Gamma \vdash \mathtt{t.l}_j : \mathtt{T}_j} \qquad \text{(T-PROJ)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_1 : \mathtt{T}_1} \qquad \text{(T-COERCE)}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \qquad \text{(T-FIX)}$$

$$\Gamma \vdash \mathtt{inert}\ \mathtt{[T]} : \mathtt{T} \qquad \text{(T-INERT)}$$

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

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \qquad \text{(T-TRUE)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-IF)}$$

$$\Gamma \vdash \mathtt{unit} : \mathtt{Unit} \qquad \text{(T-UNIT)}$$

$$\frac{\Sigma(l) = \mathtt{T}}{\Gamma, \Sigma \vdash l : \mathtt{Ref}\ \mathtt{T}} \qquad \text{(T-LOC)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma, \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-REF)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1}{\Gamma, \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_1} \qquad \text{(T-DEREF)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1 \qquad \Gamma, \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma, \Sigma \vdash \mathtt{t}_1\mathtt{:=t}_2 : \mathtt{Unit}} \qquad \text{(T-ASSIGN)}$$

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ \mathtt{t_1} : \mathtt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{Nat}}{\Gamma \vdash \mathtt{iszero}\ \mathtt{t_1} : \mathtt{Bool}} \qquad \text{(T-ISZERO)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T_2}.\mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}{\to}\mathtt{T_1} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_1}} \qquad \text{(T-APP)}$$

Abbreviations:

$$\mathtt{\{...t_i...\}} \stackrel{\text{def}}{=} \mathtt{\{...i=t_i...\}}$$

$$\mathtt{\{...T_i...\}} \stackrel{\text{def}}{=} \mathtt{\{...i{:}T_i...\}}$$

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

8.1.2 Properties

8.1.2.1 Definition: We say that a store μ is well-typed with respect to a typing context Γ and a store typing Σ, written $\Gamma, \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma, \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$. □

8.1.2.2 Lemma [Substitution]: If $\Gamma, \mathtt{x}{:}\mathtt{S}, \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma, \Sigma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma, \Sigma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t} : \mathtt{T}$. □

Proof: Just like the proof of Lemma 6.4.2.1. □

8.1.2.3 Lemma: If $\Gamma, \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\Sigma' \supseteq \Sigma$, then $\Gamma, \Sigma' \vdash \mathtt{t} : \mathtt{T}$. □

Proof: Easy. □

8.1.2.4 Theorem [Subject reduction]: If

$$\Gamma, \Sigma \vdash \mathtt{t} : \mathtt{T}$$
$$\Gamma, \Sigma \vdash \mu$$
$$\mathtt{t} \mid \mu \longrightarrow^* \mathtt{t'} \mid \mu'$$

then, for some $\Sigma' \supseteq \Sigma$,

$$\Gamma, \Sigma' \vdash \mathtt{t'} : \mathtt{T}$$
$$\Gamma, \Sigma' \vdash \mu'.$$

□

Proof: Clearly, the subject reduction property for multi-step evaluation will follow (by a straightforward induction) from subject reduction for single-step evaluation. This, in turn, is established by an induction on a derivation of $\mathtt{t} \mid \mu \longrightarrow^{*} \mathtt{t'} \mid \mu'$. All the cases of this induction are straightforward, using the lemmas above and the evident inversion property of the typing rules. □

8.1.2.5 Exercise: Is the evaluation relation defined in this chapter strongly normalizing on well-typed terms? (Solution on page **??**.) □

8.2 Further Reading

The presentation of references in this chapter is adapted from Harper's treatment **??**. An earlier account in a similar style is given by Wright and Felleisen [**?**].

The combination of references (or other computational effects) with ML-style polymorphic type inference (cf. Section **??**) raises some subtle problems and has received a good deal of attention in the research literature. See [Tof90, HMV93, JG91, TJ92, TJ92, LW91, Wri92] and the references cited there.

Chapter 9

Type Equivalence

$\Box$: **Extends**

New type equivalence rules:

$$\boxed{\mathtt{T} \equiv \mathtt{T}} \qquad \text{(TEQV-REFL)}$$

$$\frac{\mathtt{S} \equiv \mathtt{T}}{\mathtt{T} \equiv \mathtt{S}} \qquad \text{(TEQV-SYMM)}$$

$$\frac{\mathtt{S} \equiv \mathtt{T} \qquad \mathtt{T} \equiv \mathtt{U}}{\mathtt{S} \equiv \mathtt{U}} \qquad \text{(TEQV-TRANS)}$$

$$\frac{\pi \text{ is a permutation of } \{1..n\}}{\{\mathtt{l}_i\!:\!\mathtt{T}_i{}^{i\in 1..n}\} \equiv \{\mathtt{l}_{\pi i}\!:\!\mathtt{T}_{\pi i}^{i\in 1..n}\}} \qquad \text{(TEQV-RCD-PERM)}$$

New typing rules:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{S} \qquad \mathtt{S} \equiv \mathtt{T}}{\Gamma \vdash \mathtt{s} : \mathtt{T}} \qquad \text{(T-EQ)}$$

Chapter 10

Subtyping

In programming in the simply typed lambda-calculus, we may sometimes find ourselves irritated by the type system's insistence that types match exactly, for example in the rule T-APP for typing applications:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{(T-APP)}$$

According to this rule, the term

```
(λr:{x:Nat}. r.x) {x=0, y=1}
```

is not typeable, even though it will obviously evaluate without producing any run-time errors, since the function only requires that its argument has a field `x`; it doesn't care what other fields the argument may or may not have. Moreover, this fact is evident from the type of the function—we don't need to look at its body to see that it doesn't use any fields besides `x`.

One way to formalize this observation is to extend the typing rules so that the term `{x=0, y=0}` is given a *set* of types including both `{x:Nat, y:Nat}` and `{x:Nat}`. Then the application above is well typed because one of the possible types of the argument matches the left-hand side of the type of the function.

To accomplish this in a general way, we introduce a **principle of safe substitution**: We say that "`S` is a subtype of `T`," written `S <: T`, to mean that any term of type `S` can safely be used in a context where a term of type `T` is required. If this is the case, then a term `t` of type `S` is also given the type `T`, using the so-called rule of **subsumption**:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{s} : \mathtt{T}} \qquad \text{(T-SUB)}$$

No other changes are needed to the typing relation (though, as we shall see in Section 10.3, some changes are required to the algorithmic *implementation* of the typing relation). It remains only to formalize the subtype relation.

In this chapter, we'll mainly work with a calculus with just records and functions (written $\lambda_{<:\{\}}$, for short), which is rich enough to bring out most of the interesting issues. Examples and exercises will touch on the combination of subtyping with most of the other features we have seen.

10.1 The Subtype Relation

Clearly, it is not safe to substitute functions for records, records for functions, etc. The subtyping relation will only be defined for types with similar structure.

First, just to make sure that our subtyping relation is sensible, we make two general stipulations: first, that it should be reflexive, and second, that it should be transitive:

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \tag{S-Trans}$$

These rules follow directly from the intuition of safe substitutivity. The remainder of the rules defining the subtype relation concern the behavior of specific type constructors.

For record types, we have already seen that we want to consider the type $\mathtt{S} = \{\mathtt{k}_1\mathtt{:S}_1\ldots\mathtt{k}_m\mathtt{:S}_m\}$ to be a subtype of $\mathtt{T} = \{\mathtt{l}_1\mathtt{:T}_1\ldots\mathtt{l}_n\mathtt{:T}_n\}$ if $\mathtt{T}$ is obtained by dropping fields from $\mathtt{S}$.

$$\{\mathtt{l}_i\mathtt{:T}_i{}^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i{}^{i\in 1..n}\} \tag{S-Rcd-Width}$$

In fact, we can be a little more general than this and also allow the types of the common fields to differ, so long as their types in $\mathtt{S}$ are subtypes of their types in $\mathtt{T}$.

$$\frac{\mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i{}^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i{}^{i\in 1..n}\}} \tag{S-Rcd-Depth}$$

Also, it makes sense to ignore the order of fields in a record, since the only thing that we can *do* with records—namely, field projection—is insensitive to the order of fields.

For functions, the rule is a little trickier:

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1\to\mathtt{S}_2 <: \mathtt{T}_1\to\mathtt{T}_2} \tag{S-Arrow}$$

That is, we allow a function of one type to be used where another is expected as long as none of the arguments that may be passed to the function by the context will surprise it ($\mathtt{T}_1 <: \mathtt{S}_1$) and none of the results that it returns will surprise the context ($\mathtt{S}_2 <: \mathtt{T}_2$). Notice that the sense of the subtype relation is reversed on the left of the arrow; this rule is sometimes referred to as the "contravariant arrow rule" for this reason.

10.1.1 Exercise [Quick check]: Demonstrate that the premises of S-Arrow are both necessary by giving ill-behaved programs that would be well typed if either premise were dropped. □

Finally, for various reasons, it will be convenient to have a type that is a supertype of every type. We introduce a new type constant `Top` for this purpose, plus a rule that makes `Top` maximal in the subtype relation.

$$\texttt{S} <: \texttt{Top} \qquad \text{(S-TOP)}$$

Formally, the subtype relation is the least relation closed under the rules we have given.

10.1.2 Exercise [Quick check]: In a system with reference cells, would it be a good idea to add a rule

$$\frac{\texttt{S} <: \texttt{T}}{\texttt{Ref S} <: \texttt{Ref T}}$$

for subtyping between reference cell types? □

10.1.1 Variance

We pause for a moment to introduce some useful terminology for discussing types and subtyping.

Suppose `T[]` is a type term containing some number of holes, written "[]" someplace inside it; we write `T(S)` for the term that results from filling the holes in `T` with `S`. For example, if $\texttt{T[]} = \texttt{\{x:Top,y:[]\}}\to\texttt{Top}\to\texttt{[]}\to\texttt{Nat}$, then $\texttt{T(Nat)} = \texttt{\{x:Top,y:Nat\}}\to\texttt{Top}\to\texttt{Nat}\to\texttt{Nat}$.

We say that `T[]` is **covariant** in the position(s) marked by the holes if, whenever $\texttt{S}_1 <: \texttt{S}_2$, we have $\texttt{T(S}_1\texttt{)} <: \texttt{T(S}_2\texttt{)}$. We say that `T[]` is **contravariant** if, whenever $\texttt{S}_1 <: \texttt{S}_2$, we have $\texttt{T(S}_2\texttt{)} <: \texttt{T(S}_1\texttt{)}$. If `T[]` is neither covariant nor contravariant, it is said to be **invariant**. For example, we say that the arrow type constructor is contravariant in its first argument and covariant in its second, since $\texttt{[]}\to\texttt{S}$ is contravariant while $\texttt{S}\to\texttt{[]}$ is covariant.

10.1.1.1 Exercise: Which of the following types are covariant in the position marked by the hole(s)? Which are contravariant? Which are invariant?

1. $\texttt{T[]} = \texttt{[]}\to\texttt{Top}$
2. $\texttt{T[]} = \texttt{Top}\to\texttt{[]}$
3. $\texttt{T[]} = \texttt{\{x:Top,y:[]\}}\to\texttt{Nat}\to\texttt{Nat}$
4. $\texttt{T[]} = \texttt{[]}\to\texttt{Top}\to\texttt{Top}$
5. $\texttt{T[]} = \texttt{([]}\to\texttt{Top)}\to\texttt{Top}$
6. $\texttt{T[]} = \texttt{[]}\to\texttt{Nat}\to\texttt{[]}$
7. $\texttt{T[]} = \texttt{[]}$
8. $\texttt{T[]} = \texttt{Nat}$ □

10.1.2 Summary

As usual, we present the subtyping extension of the pure simply typed lambda calculus first, then the additional rules for records.

□ : **Extends**

Syntax:

t	::=	x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=	x	*variable*
		λx:T.t	*abstraction value*
T	::=	$\boxed{\texttt{Top}}$	*maximum type*
		T→T	*type of functions*
Γ	::=	∅	*empty context*
		Γ, x:T	*term variable binding*

Evaluation:

$$(\lambda x{:}T_2.t_1)\ v_2 \longrightarrow \{x \mapsto v_2\}t_1 \qquad \text{(EV-BETA)}$$

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(EV-APP1)}$$

$$\frac{t_2 \longrightarrow t_2{}'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(EV-APP2)}$$

Subtyping:

$$\boxed{S <: S} \qquad \text{(S-REFL)}$$

$$\boxed{\frac{S <: U \qquad U <: T}{S <: T}} \qquad \text{(S-TRANS)}$$

$$\boxed{S <: \texttt{Top}} \qquad \text{(S-TOP)}$$

$$\boxed{\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}} \qquad \text{(S-ARROW)}$$

Typing:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{s} : \mathtt{T}} \qquad \text{(T-SUB)}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{(T-APP)}$$

$\square$: **Extends**

New type equivalence rules:

$$\mathtt{T} \equiv \mathtt{T} \qquad \text{(TEQV-REFL)}$$

$$\frac{\mathtt{S} \equiv \mathtt{T}}{\mathtt{T} \equiv \mathtt{S}} \qquad \text{(TEQV-SYMM)}$$

$$\frac{\mathtt{S} \equiv \mathtt{T} \qquad \mathtt{T} \equiv \mathtt{U}}{\mathtt{S} \equiv \mathtt{U}} \qquad \text{(TEQV-TRANS)}$$

$$\frac{\pi \text{ is a permutation of } \{1..n\}}{\{\mathtt{l}_i{:}\mathtt{T}_i{}^{i\in 1..n}\} \equiv \{\mathtt{l}_{\pi i}{:}\mathtt{T}_{\pi i}^{i\in 1..n}\}} \qquad \text{(TEQV-RCD-PERM)}$$

New subtyping rules:

$$\frac{\mathtt{S} \equiv \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \qquad \text{(S-EQV)}$$

$$\{\mathtt{l}_i{:}\mathtt{T}_i{}^{i\in 1..n+k}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i{}^{i\in 1..n}\} \qquad \text{(S-RCD-WIDTH)}$$

$$\frac{\mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i{:}\mathtt{S}_i{}^{i\in 1..n}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i{}^{i\in 1..n}\}} \qquad \text{(S-RCD-DEPTH)}$$

New typing rules:

10.1.2.1 Exercise: What should the subtyping rules for variants be? □

10.1.2.2 Exercise: In what ways are the `Top` type and the empty record type `{}` similar? In what ways are they different? □

10.2 Extensions

10.2.1 Primitive Subtyping

10.2.2 The Bottom Type

10.3 Metatheory of Subtyping

In this section we consider some of the properties of the system $\lambda_{\rightarrow\{\}<:}$, the simply typed lambda-calculus with records and subtyping.

10.3.1 Algorithmic Subtyping

The three rules for record subtyping are often combined into one more compact but less readable rule:

$$\frac{\{\mathtt{l}_i{}^{i\in 1..n}\} \subseteq \{\mathtt{k}_j{}^{j\in 1..m}\} \qquad \mathtt{k}_j = \mathtt{l}_i \text{ implies } \mathtt{S}_j <: \mathtt{T}_i}{\{\mathtt{k}_j{:}\mathtt{S}_j{}^{j\in 1..m}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i{}^{i\in 1..n}\}} \tag{S-RCD}$$

10.3.1.1 Lemma: This rule can be derived from S-RCD-DEPTH, S-RECORD-WIDTH, and S-RECORD-PERM (using reflexivity and transitivity) and vice versa. □

Proof: Exercise. □

We now observe that the rules of reflexivity and transitivity are inessential, in the following sense:

10.3.1.2 Lemma:

1. $\mathtt{S} <: \mathtt{S}$ can be derived for every type `S` without using S-REFL.
2. If $\mathtt{S} <: \mathtt{T}$ can be derived at all, then it can be derived without using S-TRANS. □

Proof: Exercise. □

10.3.1.3 Exercise [Quick check]: If we add `Nat` to the type system, how do these properties change? □

10.3.1.4 Definition: The **algorithmic subtyping** relation is the least relation closed under the following rules:

$$\mathtt{S} <:_a \mathtt{Top} \qquad \text{(SA-TOP)}$$

$$\frac{\mathtt{T}_1 <:_a \mathtt{S}_1 \qquad \mathtt{S}_2 <:_a \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 <:_a \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(SA-ARROW)}$$

$$\frac{\{\mathtt{l}_i\ ^{i \in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j \in 1..m}\} \qquad \mathtt{k}_j = \mathtt{l}_i \text{ implies } \mathtt{S}_j <:_a \mathtt{T}_i}{\{\mathtt{k}_j : \mathtt{S}_j\ ^{j \in 1..m}\} <:_a \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}} \qquad \text{(SA-RCD)}$$

□

10.3.1.5 Proposition [Soundness and completeness of algorithmic subtyping]: $\mathtt{S} <: \mathtt{T}$ iff $\mathtt{S} <:_a \mathtt{T}$. □

Proof: Using the two previous lemmas. □

This, in turn, leads directly to an algorithm for checking the algorithmic subtype relation (and hence also the declarative subtype relation, by Proposition 10.3.1.5).

$$
\begin{array}{lll}
\mathit{subtype}(\mathtt{S},\mathtt{T}) & = & \text{if } \mathtt{T} = \mathtt{Top}, \\
 & & \quad \text{then } \mathit{true} \\
 & & \text{else if } \mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2 \\
 & & \quad \text{then } \mathit{subtype}(\mathtt{T}_1,\mathtt{S}_1) \ \wedge\ \mathit{subtype}(\mathtt{S}_2,\mathtt{T}_2) \\
 & & \text{else if } \mathtt{S} = \{\mathtt{k}_j : \mathtt{S}_j\ ^{j \in 1..m}\} \text{ and } \mathtt{T} = \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\} \\
 & & \quad \text{then, for all } i \in 1..n, \\
 & & \qquad \text{find } j \text{ such that } \mathtt{k}_j = \mathtt{l}_i \text{ and check} \\
 & & \qquad\quad \mathit{subtype}(\mathtt{S}_j,\mathtt{T}_i) \\
 & & \text{else} \\
 & & \quad \mathit{false}.
\end{array}
$$

10.3.2 Minimal Typing

By introducing the rule of subsumption into the typing relation, we have lost the useful **syntax-directedness** property, which in the simply typed lambda-calculus allows a typechecker to be implemented simply by "reading the rules from bottom to top." Without this property, how can we now check whether $\Gamma \vdash \mathtt{t} : \mathtt{T}$ is a derivable statement?

We do it in two steps:

1. Find another set of syntax rules that *are* syntax directed, defining a relation $\Gamma \vdash^a \mathtt{t} : \mathtt{T}$. This is called the **algorithmic typing relation**.

2. Show that, although they are defined differently, the ordinary typing relation $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and the algorithmic typing relation $\Gamma \vdash^a \mathtt{t} : \mathtt{T}$ give rise to the same set of typable terms. Interestingly, though, they will *not* assign exactly the same types to those terms.

In order to find an algorithmic presentation of the typing relation, we need to think carefully about the structure of typing derivations to see where the rule of subsumption is really needed, and where we could just as well do without it.

For example, consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}} \qquad \dfrac{\vdots}{\mathtt{T} <: \mathtt{T}'}}{\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}'}\;(\text{T-Sub})}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t} : \mathtt{S}\to\mathtt{T}'}\;(\text{T-Abs})$$

Any such derivation can be rearranged like this:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t} : \mathtt{S}\to\mathtt{T}}\;(\text{T-Abs}) \qquad \dfrac{\dfrac{}{\mathtt{S} <: \mathtt{S}}\;(\text{S-Refl}) \qquad \dfrac{\vdots}{\mathtt{T} <: \mathtt{T}'}}{\mathtt{S}\to\mathtt{T} <: \mathtt{S}\to\mathtt{T}'}\;(\text{S-Abs})}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t} : \mathtt{S}\to\mathtt{T}'}\;(\text{T-Sub})$$

So we can get away with not allowing instances of T-SUB as the final rule in the right-hand premise of an instance of T-ABS.

Similarly, any derivation of the form

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}_1\to\mathtt{S}_2} \qquad \dfrac{\dfrac{\vdots}{\mathtt{T}_1 <: \mathtt{S}_1} \qquad \dfrac{\vdots}{\mathtt{S}_2 <: \mathtt{T}_2}}{\mathtt{S}_1\to\mathtt{S}_2 <: \mathtt{T}_1\to\mathtt{T}_2}\;(\text{S-Arrow})}{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1\to\mathtt{T}_2}\;(\text{T-Sub}) \qquad \dfrac{\vdots}{\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_2}$$

can be replaced by one of the form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}_1\to\mathtt{S}_2} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1} \qquad \dfrac{\vdots}{\mathtt{T}_1 <: \mathtt{S}_1}}{\Gamma \vdash \mathtt{t}_2 : \mathtt{S}_1}\;(\text{T-Sub})}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{S}_2}\;(\text{T-App}) \qquad \dfrac{\vdots}{\mathtt{S}_2 <: \mathtt{T}_2}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_2}\;(\text{T-Sub})$$

That is, we can take any derivation where subsumption is used as the final rule in the left-hand premise of an instance of T-APP and replace it by a derivation with one instance of subsumption appearing at the end of the right-hand premise and another instance of subsumption appearing at the end of the whole derivation.

Finally, adjacent uses of subsumption can be coalesced, in the sense that any derivation of the form

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma \vdash \mathtt{t} : \mathtt{S}} \qquad \cfrac{\vdots}{\mathtt{S} <: \mathtt{U}}}{\Gamma \vdash \mathtt{t} : \mathtt{U}}\ (\text{T-Sub}) \qquad \cfrac{\vdots}{\mathtt{U} <: \mathtt{T}}}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\ (\text{T-Sub})$$

can be rewritten:

$$\cfrac{\cfrac{\vdots}{\Gamma \vdash \mathtt{t} : \mathtt{S}} \qquad \cfrac{\cfrac{\vdots}{\mathtt{S} <: \mathtt{U}} \qquad \cfrac{\vdots}{\mathtt{U} <: \mathtt{T}}}{\mathtt{S} <: \mathtt{T}}\ (\text{S-Trans})}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\ (\text{T-Sub})$$

10.3.2.1 Exercise [Recommended]: Show that a similar rearrangement can be performed on derivations in which T-SUB is used immediately before T-PROJ. □

A motivating intuition behind all of these transformations is that nothing is harmed by moving most uses of subsumption as late as possible in a typing derivation, so that each subterm is given the smallest possible (or **minimal**) type at the outset and this small type is maintained until a point is reached (in the application rule) where a larger type is needed. This motivates the following definition:

10.3.2.2 Definition: The **algorithmic typing relation** is the least relation closed under the following rules:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash^a \mathtt{x} : \mathtt{T}} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T_1} \vdash^a \mathtt{t_1} : \mathtt{T_2} \qquad \mathtt{x} \notin \mathit{dom}(\Gamma)}{\Gamma \vdash^a \lambda\mathtt{x}{:}\mathtt{T_1}.\mathtt{t_1} : \mathtt{T_1} \to \mathtt{T_2}} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \vdash^a \mathtt{t_1} : \mathtt{T_1} \to \mathtt{T_2} \qquad \Gamma \vdash^a \mathtt{t_2} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T_1}}{\Gamma \vdash^a \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_2}} \qquad \text{(TA-APP)}$$

$$\frac{\Gamma \vdash^a \mathtt{t_i} : \mathtt{T_i} \quad \text{for each } i}{\Gamma \vdash^a \{\mathtt{l_1}{=}\mathtt{t_1} \ldots \mathtt{l_n}{=}\mathtt{t_n}\} : \{\mathtt{l_1}{:}\mathtt{T_1} \ldots \mathtt{l_n}{:}\mathtt{T_n}\}} \qquad \text{(TA-RCD)}$$

$$\frac{\Gamma \vdash^a \mathtt{t} : \{\mathtt{l_1}{:}\mathtt{T_1} \ldots \mathtt{l_n}{:}\mathtt{T_n}\}}{\Gamma \vdash^a \mathtt{t}.\mathtt{l_i} : \mathtt{T_i}} \qquad \text{(TA-DOT)}$$

□

10.3.2.3 Exercise [Recommended]: Show, by example, that minimal types of terms in this calculus can *decrease* during evaluation. Give a term `t` such that $\texttt{t} \longrightarrow^* \texttt{r}$ and `R <: T` but $\texttt{T} \not<: \texttt{R}$, where `R` is the minimal type of `r` and `T` is the minimal type of `t`. □

We now want to see formally that the algorithmic typing rules define the same relation as the ordinary rules. We do this in two steps:

Soundness: Every typing statement that can be proved by an algorithmic derivation can also be proved by an ordinary derivation.

Completeness: Every typing statement that can be proved by an ordinary derivation can *almost* be proved by an algorithmic derivation. (In fact, the algorithmic derivation may yield a better type.)

10.3.2.4 Theorem [Soundness of the algorithmic typing relation]: If $\Gamma \vdash^a \texttt{t : T}$, then $\Gamma \vdash \texttt{t : T}$. □

Proof: By straightforward induction on algorithmic typing derivations. The details are left as an exercise. □

The ordinary typing relation can be used to assign many types to a term, while the algorithmic typing relation assigns at most one (this is easy to check). So a straightforward converse of Theorem 10.3.2.4 is clearly not going to hold. Instead, we can show that if a term `t` has a type `T` under the ordinary typing rules, then it has a *better* type `S` under the algorithmic rules, in the sense that `S <: T`. In other words, the algorithmic rules assign each typeable term its smallest possible ("minimal") type.

10.3.2.5 Theorem [Minimal Typing]: If $\Gamma \vdash \texttt{t : T}$, then $\Gamma \vdash^a \texttt{t : S}$ for some `S <: T`. □

Proof: Exercise (recommended). □

10.3.2.6 Exercise: If we dropped the function subtyping rule S-ARROW but kept the rest of the subtyping and typing rules the same, would we lose the minimal typing property? If not, prove it. If so, give an example of a term that would have two incomparable types. □

10.3.2.7 Exercise [Recommended]: Add numbers and iteration to the algorithmic typing relation. Prove that your algorithmic rules are sound and complete for the declarative system. □

10.3.2.8 Exercise [Recommended]: Show how to extend the proof of subject reduction for the simply typed lambda-calculus (**??**) to the system with subtyping. □

10.4 Implementation

```
let rec subtype tyS tyT =
   tyeqv tyS tyT ||
   match (tyS,tyT) with
     (TyRecord(fS), TyRecord(fT)) →
       List.for_all
         (fun (li,tyTi) →
            try let tySi = List.assoc li fS in
                subtype tySi tyTi
            with Not_found → false)
         fT
   | (_,TyTop) →
       true
   | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
       (subtype tyT1 tyS1) && (subtype tyS2 tyT2)
   | _ →
       false
```

```
let rec typeof ctx t =
  match t with
    TmRecord(fi, fields) →
      let fieldtys = List.map (fun (li,ti) → (li, typeof ctx ti)) fields in
      TyRecord(fieldtys)
  | TmProj(fi, t1, l) →
      (match (typeof ctx t1) with
          TyRecord(fieldtys) →
            (try List.assoc l fieldtys
             with Not_found → error fi ("label "^l^" not present"))
        | _ → error fi "Expected record type")
  | TmVar(fi,i,_) →
      gettype fi ctx i
  | TmAbs(fi,x,tyS,t1) →
      let ctx' = addbinding ctx x (VarBind(tyS)) in
      let tyT = typeof ctx' t1 in
      TyArr(tyS, tyshift tyT (-1))
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if subtype tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
```

10.5 Meets and Joins

A type J is called a **join** of a pair of types S and T if

> ⊢ S <: J
> ⊢ T <: J
> for all types U, if ⊢ S <: U and ⊢ T <: U then ⊢ J <: U.

A given typed lambda-calculus is said to **have joins** if, for every S and T, there is some J that is a join of S and T under Γ.

Similarly, a type M is called a **meet** of S and T if

> ⊢ M <: S
> ⊢ M <: T
> for all types L, if ⊢ L <: S and ⊢ L <: T then ⊢ L <: M.

A pair of types S and T is said to be **bounded below** if there is some type L such that ⊢ L <: S and ⊢ L <: T. A typed lambda-calculus is said to **have bounded meets** if, for every S and T such that S and T are bounded below, there is some M that is a meet of S and T.

```
let rec join tyS tyT =
  if subtype tyS tyT then tyT
  else if subtype tyT tyS then tyS
  else match (tyS,tyT) with
        (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
           (try TyArr(meet tyS1 tyT1, join tyS2 tyT2)
            with Not_found → TyTop)
      | (TyRecord(f1), TyRecord(f2)) →
          let rec loop = function
              (_,[]) → []
            | ([],_) → []
            | ((l1,t1)::rest1,(l2,t2)::rest2) →
               if l1 = l2 then
                 (l1,(join t1 t2))::(loop (rest1,rest2))
               else []
          in let f = loop (f1,f2)
          in (match f with
                (* when the first field names did not agree *)
                [] → TyTop
              | _ → TyRecord(f))
      | _ → TyTop

and meet tyS tyT =
```

```
  if subtype tyS tyT then tyS
  else if subtype tyT tyS then tyT
  else
    match (tyS,tyT) with
        (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
           TyArr(join tyS1 tyT1, meet tyS2 tyT2)
      | (TyRecord(f1), TyRecord(f2)) →
          let rec loop = function
              (restf1,[]) → restf1
            | ([],restf2) → restf2
            | ((l1,t1)::rest1,(l2,t2)::rest2) →
               if l1 = l2 then
                 (l1,(meet t1 t2))::(loop (rest1,rest2))
               else raise Not_found
          in let f = loop (f1,f2)
          in TyRecord(f)
      | _ → raise Not_found

let rec typeof ctx t =
  match t with
    ...
  | TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
     TyBool
  | TmIf(fi,s1,s2,s3) →
      if subtype (typeof ctx s1) TyBool then
        join (typeof ctx s2) (typeof ctx s3)
      else error fi "guard of conditional not a boolean"
```

10.5.1 Exercise [Recommended]: Add booleans and conditionals to both the declarative typing relation and the algorithmic typing relation. Prove that your algorithmic rules are sound and complete for the declarative system. □

10.5.2 Exercise [Recommended]: The directory `simpleplus` in the course web site contains an ML checker for the simply typed lambda-calculus with records, numbers, and booleans. Add subtyping (and a `Top` type) to this checker using the algorithms described in this chapter. □

Chapter 11

Imperative Objects

In this chapter we come to our first substantial programming examples. We'll use the features we have defined – functions, records, effects, fixed points, reference cells, and subtyping – to give a straightforward model of objects and classes, as they are found in object-oriented languages such as Smalltalk and Java. We begin with very simple "stand-alone" objects and then proceed to define increasingly powerful kinds of classes.

11.1 Objects

An **object** is a data structure encapsulating some internal **state** and offering access to this state to clients via a collection of **methods**. The internal state is typically broken up into a number of mutable **instance variables** (or **fields**) that are shared among the definitions of the methods (and typically inaccessible to the rest of the program). An object's methods are invoked by the operation of **message-sending**.

Our running example for the chapter will be simple "storage cell" objects. Each cell object will hold a single number and respond to three messages: `get`, which causes it to return the number it currently has; `set`, which sets its internal state to some number given by the sender of the message; and `inc`, which causes it to increment the number it is holding. (Except for the `inc` message, our cells are just an object-oriented variation of the `refs` that we saw in Chapter 8.)

A very simple way of obtaining this behavior using the features we have studied so far is to use a reference cell for the object's internal state and a record of functions for the methods.

```
c =
  let r = ref 0 in
    {get = λ_:Unit. !r,
     set = λi:Nat. r:=i,
```

```
       inc = λ_:Unit. r:=succ(!r)};
▶ c : {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit}
```

The fact that the state is shared by the methods and inaccessible to the rest of the program arises directly from the lexical scope of the variable `r`. "Sending a message" to the object `c` means just extracting some field of the record and applying it to an appropriate argument. For example:

```
  c.set 2;
  c.inc unit;
  c.get unit;
  (c.set 2; c.inc unit; c.get unit);
▶ unit : Unit
  unit : Unit
  3 : Nat
  3 : Nat
```

Note the use of the syntactic sugar for sequencing (cf. Section **??**) in the last line. We could equivalently (but less readably) have written

```
let x = c.set 2 in let y = c.inc unit in c.get unit;
```

or even (using the _ convention for "don't care" variable names):

```
let _ = c.set 2 in let _ = c.inc unit in c.get unit;
```

We may want to create and manipulate a lot of counters, so it will be convenient to use an abbreviation for their rather long record type:

```
Counter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

To force the typechecker to print the type of a new counter using the short form `Counter`, we add an explicit coercion (recall from Section **??** that `t as T` means the same as `(λx:T.x)t`):

```
  c =
    let r = ref 0 in
      {get = λ_:Unit. !r,
       set = λi:Nat. r:=i,
       inc = λ_:Unit. r:=succ(!r)}
    as Counter;
▶ c : Counter
```

(This need to explicitly annotate terms to control the way their types are printed is a little annoying. It is the price we pay for "unbundling" all the features of our programming language into the simplest and most orthogonal possible units, so that we can study them in isolation. A practical high-level programming language design would probably combine many of these primitive features into more convenient macro-constructs.)

A typical piece of "client code" that manipulates counter objects might look something like this:

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
```

```
▸ inc3 : Counter → Unit
```

The function `inc3` takes a counter object and increments it three times by sending it three `inc` messages in succession.

```
(c.set 5; inc3 c; c.get unit);
```

```
▸ 8 : Nat
```

11.2 Object Generators

In the development that follows, it will be useful to be able to manipulate all the instance variables of an object as a single unit. To support this, let us change the internal representation of our counters to be a *record* of reference cells and use the name of this record to refer to instance variables from the method body.

```
c =
  let r = {x=ref 0} in
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. r.x:=succ(!(r.x))}
  as Counter;
```

```
▸ c : Counter
```

This representation makes it is easy, for example, to write a *counter generator* that creates a new counter every time it is called.

```
newCounter =
  λ_:Unit.
  (let r = {x=ref 0} in
     {get = λ_:Unit. !(r.x),
      set = λi:Nat. r.x:=i,
      inc = λ_:Unit. r.x:=succ(!(r.x))}
   as Counter);
```

```
▸ newCounter : Unit → Counter
```

11.3 Subtyping

One of the reasons for the popularity of object-oriented programming styles is that they permit objects of many shapes to be manipulated by the same client code.

For example, suppose that, in addition to the `Counter` objects defined above, we also create some cell objects with one more method that allows them to be reset to their initial state (say, 0) at any time.

```
ResetCounter = {get:Unit→Nat,
                     set:Nat→Unit,
                     inc:Unit→Unit,
                     reset:Unit→Unit};

newResetCounter =
  λ_:Unit.
    let r = {x=ref 0} in
       {get   = λ_:Unit. !(r.x),
        set   = λi:Nat. r.x:=i,
        inc   = λ_:Unit. r.x:=succ(!(r.x)),
        reset = λ_:Unit. r.x:=0}
    as ResetCounter;
▶ newResetCounter : Unit → ResetCounter
```

Note that our old `inc3` function on counters can safely use our refined counters too:

```
rc = newResetCounter unit;
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
▶ rc : ResetCounter
3 : Nat
```

11.4 Basic classes

The definitions of `newCounter` and `newResetCounter` are identical except for the body of the `reset` method. Of course, for such short definitions this makes little difference, but if the method bodies were much longer we might quickly find ourselves wanting to write the common ones in just one place. The mechanism by which this is achieved in most object-oriented languages is called the **class**.

Here is one very simple way to define `newResetCounter` in terms of `newCounter`:

```
newResetCounter =
  λ_:Unit.
  (let c = newCounter unit in
   {get   = c.get,
    set   = c.set,
    inc   = c.inc,
    reset = λ_:Unit. c.set 0}
   as ResetCounter);
```

```
▶ newResetCounter : Unit → ResetCounter
```

But this trick is not very robust. For example, if the type `Counter` had just `get` and `inc` methods (no `set`), then we couldn't write `newResetCounter` in this way.

A more general approach is to abstract the methods of a prototype counter object over the instance variables. This is a simple example of a **class**.

```
CounterRep = {x: Ref Nat};
counterClass =
  λr:CounterRep.
    ({get = λ_:Unit. !(r.x),
      set = λi:Nat. r.x:=i,
      inc = λ_:Unit. r.x:=succ(!(r.x))}
    as Counter);
```

```
▶ counterClass : CounterRep → Counter
```

Now we can build a class for `ResetCounter` by re-using the fields of `counterClass`:

```
resetCounterClass =
  λr:CounterRep.
  let super = counterClass r in
    ({get   = super.get,
      set   = super.set,
      inc   = super.inc,
      reset = λ_:Unit. r.x:=0}
    as ResetCounter);
```

```
▶ resetCounterClass : CounterRep → ResetCounter
```

The definitions of `newCounter` and `newResetCounter` become trivial: we simply allocate the instance variables and supply them to `counterClass` or `resetCounterClass`, where the real work happens:

```
newCounter =
  λ_:Unit.
    (let r = {x=ref 0} in
       counterClass r)
    as Counter;
newResetCounter =
  λ_:Unit.
    (let r = {x=ref 0} in
       resetCounterClass r)
    as ResetCounter;
```

```
▶ newCounter : Unit → Counter
  newResetCounter : Unit → ResetCounter
```

11.5 Extending the Internal State

Suppose we want to define a class of BackupCounters whose reset method resets their state to its value at the last call to a new method backup...

```
BackupCounter = {get:Unit→Nat,
                 set:Nat→Unit,
                 inc:Unit→Unit,
                 reset:Unit→Unit,
                 backup: Unit→Unit};
BackupCounterRep = {x: Ref Nat, b: Ref Nat};
```

Here is the class:

```
backupCounterClass =
  λr:BackupCounterRep.
     ({get = λ_:Unit. !(r.x),
       set = λi:Nat. r.x:=i,
       inc = λ_:Unit. r.x:=succ(!(r.x)),
       reset = λ_:Unit. r.x:=!(r.b),
       backup = λ_:Unit. r.b:=!(r.x)}
     as BackupCounter);
▶ backupCounterClass : BackupCounterRep → BackupCounter
```

The interesting point is that we can actually define backupCounterClass in terms of resetCounterClass. Note how subtyping is used in checking the definition of super. Also, note that we need to override the definition of the reset method as well as adding backup.

```
backupCounterClass =
  λr:BackupCounterRep.
  let super = resetCounterClass r in
     ({get = super.get,
       set = super.set,
       inc = super.inc,
       reset = λ_:Unit. r.x:=!(r.b),
       backup = λ_:Unit. r.b:=!(r.x)}
     as BackupCounter);
▶ backupCounterClass : BackupCounterRep → BackupCounter
```

The variable super was used above to "copy functionality" from the superclass into the new subclass.

The super variable can also be used *inside* method definitions to extend the superclass's behavior with something new. Suppose, for instance, that we want a variant of our backupCounter class in which the inc method increments not only the cell's value but also the backed-up value in the instance variable b. (Goodness knows why this behavior would be useful—it's just an example!)

```
funnyBackupCounterClass =
  λr:BackupCounterRep.
  let super = resetCounterClass r in
     ({get = super.get,
       set = super.set,
       inc = λ_:Unit.
               (super.inc unit;
                r.b := succ(!(r.b))),
       reset = λ_:Unit. r.x:=!(r.b),
       backup = λ_:Unit. r.b:=!(r.x)}
      as BackupCounter);
```

```
▶ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

Note how use of `super.inc` in the definition of `inc` avoids repeating the superclass's `inc` code here.

11.6 Classes with "Self"

Our final extension is allowing the methods of classes to refer to each other "recursively." For example, suppose that we want to implement the inc method of simple counters in terms of the get and set methods. (Of course, all three methods are so small in this example that all this code reuse is hardly worth the trouble! For larger examples, though, it can make a substantial difference in code size and maintainability.)

```
counterClass =
  λr:CounterRep.
    fix
      (λself: Counter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat. r.x:=i,
         inc = λ_:Unit. self.set (succ(self.get unit))});
```

```
▶ counterClass : CounterRep → {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit}
```

Classes in mainstream object-oriented languages such as Smalltalk, C++, and Java actually support a more general form of recursive call between methods, sometimes known as **open recursion**. Instead of taking the fixed point within the class, we wait until we use the class to actually create an object.

```
counterClass =
  λr:CounterRep.
     λself: Counter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat. r.x:=i,
        inc = λ_:Unit. self.set (succ(self.get unit))};
```

```
▶ counterClass : CounterRep →
                 Counter → {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit}

  newCounter =
    λ_:Unit.
      let r = {x=ref 0} in
      fix (counterClass r);
▶ newCounter : Unit → {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit}
```

What's interesting about leaving the recursion open is that, when we build a new class by subclassing `counterClass`, we can override the methods of `counterClass` in such a way that recursive calls to these methods *from within* `counterClass` will actually invoke the new method bodies that we provide. For example, here's a subclass that keeps track of how many times the `get` method has been called.

```
  InstrumentedCounter = {get:Unit→Nat,
                         set:Nat→Unit,
                         inc:Unit→Unit,
                         accesses:Unit→Nat};
  InstrumentedCounterRep = {x: Ref Nat, a: Ref Nat};
  instrumentedCounterClass =
    λr:InstrumentedCounterRep.
       λself: InstrumentedCounter.
       lazy let super = counterClass r self in
          lazy
            {get = λ_:Unit.
                      let result = super.get unit in
                      (r.a:=succ(!(r.a)); result),
             set = super.set,
             inc = super.inc,
             accesses = λ_:Unit. !(r.a)}
          as InstrumentedCounter;
▶ instrumentedCounterClass : InstrumentedCounterRep →
                             InstrumentedCounter → InstrumentedCounter
```

Notice that, because of the open recursion through `self`, the call to `get` from within `inc` results in the `a` field being incremented, even though the incrementing behavior of `get` is defined in the subclass and the call to `get` appears in the superclass.

11.6.1 Exercise: The two `lazy` annotations in `newInstrumentedCounter` are both essential. Check what happens when either of them is omitted. □

We create an instrumented counter in the same way as before—by taking a fixed point of the class and providing it with some newly allocated instance variables.

```
newInstrumentedCounter =
  λ_:Unit.
  let r = {x = ref 0, a = ref 0} in
  fix (instrumentedCounterClass r);
```

▶ `newInstrumentedCounter : Unit → InstrumentedCounter`

Note how the `accesses` method counts calls to both `get` and `inc`:

```
ic = newInstrumentedCounter unit;
ic.get unit;
```

▶ `0 : Nat`

```
ic.accesses unit;
```

▶ `1 : Nat`

```
ic.inc unit;
```

▶ `unit : Unit`

```
ic.get unit;
```

▶ `1 : Nat`

```
ic.accesses unit;
```

▶ `3 : Nat`

11.6.2 Exercise [Recommended]: Use the `fullsubeff` checker from the course directory to implement the following extensions to the classes above:

1. Change the definition of `instrumentedCounterClass` so that it also counts calls to `set`.
2. Extend your modified `instrumentedCounterClass` with a subclass that adds a `reset` method (as in Section 11.3).
3. Extend this subclass with yet another subclass that supports backups as well (as in Section 11.5). □

Part III

Second-Order Systems

Chapter 12

Universal Types

12.1 Motivation

As we observed at the beginning of Chapter 10, the pure simply typed lambda-calculus is a rather restrictive system in some respects, if we consider it as the basis for a programming language. Although we can enrich it with numerous additional type constructors (references, lists, etc.), primitive types (`Nat`, `Bool`, `Unit`, etc.), and convenient control constructs (`let`, `fix`, etc.), the rigidity of the "core" typing rules can make it difficult to exploit these features to full advantage. In Chapter 10, we explored one way of refining the typing relation to make it much more flexible: adding a subtyping relation and allowing types of terms to be "promoted" when matching the types of functions and their arguments.

In this chapter, we introduce another, rather different, way of adding flexibility to the core type system. To motivate this extension, note that in $\lambda_{\to}$ there are an infinite number of "doubling" functions

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x);
doubleRcd = λf:{l:Nat}→{l:Nat}. λx:{l:Nat}. f (f x);
doubleRcd = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x);
(etc.)
```

```
▶ doubleNat : (Nat→Nat) → Nat → Nat
  doubleRcd : ({l:Nat}→{l:Nat}) → {l:Nat} → {l:Nat}
  doubleRcd : ((Nat→Nat)→Nat→Nat) → (Nat→Nat) → Nat → Nat
```

each applicable to a different type of argument, but all sharing precisely the same behavior (even the same program text, modulo typing annotations). This kind of "cut and paste" programming violates a basic principle of software engineering:

> Write each piece of functionality in one place. If something has to be used many times in slightly different ways, *abstract out* the varying parts.

Here, the varying parts are the types! What we want, it seems, are facilities for "abstracting out" a type from a term and later instantiating this abstract (or "polymorphic") term by applying it to a concrete type.

The system we'll be studying in this chapter was developed (one is tempted to say "discovered") independently, in the early 1970s by a computer scientist, John Reynolds, who called it the **polymorphic lambda-calculus** [Rey74], and by a logician, Jean-Yves Girard, who called it **System F** [Gir72]. It has been used extensively as a research vehicle for foundational work on polymorphism and as the basis for numerous programming language designs. For reasons that we shall see later (in Chapter **??**), it is sometimes called the **second-order lambda-calculus**.

12.2 Varieties of Polymorphism

Type systems that allow a single piece of code to be used with multiple types are collectively known as **polymorphic** systems. Several types of polymorphism have been proposed (this way of classifying them is due to Strachey [**?**]):

> **Parametric polymorphism**, the subject of this chapter, allows a piece of code to be typechecked "generically," using type variables in place of actual types, and then instantiated with particular types as needed. We have seen this phenomenon already in Chapter **??**, and it will be the main focus of the present chapter.
>
> **Ad-hoc polymorphism** allows different behaviors at run-time when a single piece of code is used with multiple types. The most powerful form of ad-hoc polymorphism is a general `typecase` construct that allows arbitrary computation based on type information, but this is seldom seen in real languages. More common are constructs such as Java's `hasType`, which allow simple branching on run-time type tags.
>
> **Overloading** is a very simple form of ad-hoc polymorphism where only a limited collection of (usually built-in) operations are given multiple types. For example, many languages overload arithmetic operators like $+$ with multiple types such as `Int→Int→Int` and `Real→Real→Real`.
>
> The **subtype polymorphism** that we saw in Chapter 10 gives a single term many types using the rule of subsumption, allowing us to selectively "forget" information about the term's behavior.

These categories are not exclusive: different forms of polymorphism can be mixed in the same language. For example, Standard ML offers a restricted form of parametric polymorphism and simple arithmetic overloading, but not subtyping, while Java includes subtyping, overloading, and simple ad-hoc polymorphism, but not parametric polymorphism. (There are numerous proposals for adding parametric polymorphism to Java. At the time of this writing, the front-runner is probably GJ [BOSW98].)

The bare term "polymorphism" is the source of a certain amount of confusion in the programming languages literature. In the functional programming community (i.e., those who use or design languages like ML, Haskell, etc.), it always refers to parametric polymorphism. In the object-oriented programming community, it almost always refers to subtype polymorphism.

12.3 Definitions

The definition of the polymorphic lambda-calculus (λF for short) is actually a very straightforward extension of the simply typed lambda-calculus. In $\lambda\!\rightarrow$, lambda-abstraction is used to abstract terms out of terms, and application is used to supply values for the abstracted parts. Similarly, we want a mechanism for abstracting types out of terms and filling them in later—we might as well use lambda-abstraction and application as a model. To this end, we introduce a new form of lambda-abstraction

```
λX. t
```

whose parameter is a type—a kind of function that takes a type `X` as argument. Similarly, we introduce a new form of application

```
t [T]
```

in which the argument is a type expression. We call our new functions with type parameters **polymorphic functions** or **type abstractions**; the new application construct is called **type application**.

When, during evaluation, a type abstraction meets a type application, the pair forms a redex, just as in $\lambda\!\rightarrow$. We add a reduction rule

analogous to the ordinary reduction rule for abstractions and applications. For example, when the **polymorphic identity function**

```
id = λX. λx:X. x;
```

is applied to the argument `Nat`, the result is $\{X \mapsto Nat\}(\lambda x{:}X.x)$, i.e., `λx:Nat.x`, which is the identity function on natural numbers.

Finally, we need to define the *type* of a polymorphic function. Just as we had types like `Nat→Nat` for classifying ordinary functions like `λx:Nat.x`, we now need a different form of "arrow type" whose domain is a type, for classifying polymorphic functions like `id`. Notice that, for each argument `T` to which it is applied, `id` yields a function of type `T→T`; that is, the type of the result of `id` depends on the actual type that we pass it as argument. To capture this dependency, we write the type of `id` like this:

```
id : ∀X.X→X
```

The typing rules for polymorphic abstraction and application are analogous to the corresponding rules for ordinary abstraction and application.

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_1} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_1} \qquad \text{(T-TApp)}$$

Note that we include the "binding" X in the typing context used in the subderivation for t. For the moment, this binding plays no role except to keep track of the scopes of type variables and make sure that the same type variable is not added twice to the context. In later chapters, we will annotate type variable bindings in the context with information of various kinds.

In summary, here is the complete polymorphic lambda-calculus, with differences from $\lambda{\to}$ highlighted. We'll call this system λF when we want to identify it precisely.

□: **Extends**

Syntax:

t	::=	$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
		$\lambda\mathtt{X}.\mathtt{t}$	*type abstraction*
		$\mathtt{t}\ [\mathtt{T}]$	*type application*
v	::=	$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction value*
		$\lambda\mathtt{X}.\mathtt{t}$	*type abstraction value*
T	::=	$\mathtt{X}$	*type variable*
		$\mathtt{T}{\to}\mathtt{T}$	*type of functions*
		$\forall\mathtt{X}.\mathtt{T}$	*universal type*
Γ	::=	$\emptyset$	*empty context*
		$\Gamma, \mathtt{X}$	*type variable binding*
		$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*

Evaluation:

$$(\lambda\mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_1)\ \mathtt{v}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_1 \qquad \text{(Ev-Beta)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1{}'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(Ev-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2{}'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(Ev-App2)}$$

$$(\lambda\mathtt{X}.\mathtt{t}_1)\ \mathtt{T}_2 \longrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{t}_1 \qquad \text{(Ev-Beta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ T_2 \longrightarrow t_1'\ T_2} \qquad \text{(E-TAPP)}$$

Typing:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_2 \vdash t_1 : T_1}{\Gamma \vdash \lambda x{:}T_2.t_1 : T_2 \to T_1} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, X \vdash t_1 : T_1}{\Gamma \vdash \lambda X.t_1 : \forall X.T_1} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_1}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}T_1} \qquad \text{(T-TAPP)}$$

As usual, this summary defines just the "pure" polymorphic lambda-calculus, omitting other type constructors such as records, base types such as `Nat` and `Bool`, and term-language extensions such as `let` and `fix`. These extra constructs can be added straightforwardly to the pure system, and we will use them freely in the examples that follow.

12.3.1 Exercise [Quick check]: In Exercise 6.3.1, we saw that the pure simply typed lambda calculus (with no base types or type variables) is actually a trivial system containing no typeable terms. What about pure λF? □

12.4 Examples

We now turn to developing some more interesting examples of "programming with polymorphism," of several different sorts:

- To warm up, we'll start with a few small but increasingly tricky examples, showing some of the expressive power of λF.
- We'll then review the basic ideas of "ordinary" polymorphic programming with lists, trees, etc. This is the kind of programming that usually comes to mind when polymorphism is mentioned.
- The final subsection will introduce typed versions of the Church Encodings of simple algebraic datatypes like booleans, numbers, and lists that we saw in Chapter **??** for the untyped lambda-calculus. Although these encodings are of little practical importance, they make excellent exercises for understanding the intricacies of λF.

12.4.1 Warm-ups

We have seen already how type abstraction and application can be used to define a single polymorphic identity function

```
id = λX. λx:X. x;
```

and instantiate it to yield any particular concrete identity function that may be required:

```
  id [Nat];
▶ id : ∀X. X → X
  (λx:Nat. x) : Nat → Nat

  id [Nat] 0;
▶ 0 : Nat
```

A more useful example is the polymorphic "double" function, which takes a function `f` and an argument `a` and applies `f` twice in succession to `a`:

```
  double = λX. λf:X→X. λa:X. f (f a);
▶ double : ∀X. (X→X) → X → X
```

The abstraction on the type `X` allows us to obtain doubling functions for specific types by instantiating `double` with different type arguments:

```
  doubleNat = double [Nat];
  doubleNatArrowNat = double [Nat→Nat];
▶ doubleNat : (Nat→Nat) → Nat → Nat
  doubleNatArrowNat : ((Nat→Nat)→Nat→Nat) → (Nat→Nat) → Nat → Nat
```

Once instantiated with a type argument, `double` can be further applied to an actual function and an argument of appropriate types:

```
  double [Nat] (λx:Nat. succ(succ(x))) 3;
▶ 7 : Nat
```

Here is a slightly trickier example: polymorphic self-application. Recall that, in the simply typed lambda-calculus, there is no way to give a typing to an untyped term of the form `x x` (cf. Exercise 6.4.1.5). In λF, on the other hand, this term becomes typeable if we make `x` a polymorphic function and add a type application:

```
  selfApp = λx:∀X.X→X.  x [∀X.X→X] x;
▶ selfApp : (∀X. X→X) → (∀X. X → X)
```

Here is a more useful example of self application. We can apply the polymorphic `double` function to itself, yielding a polymorphic quadrupling function:

```
  quadruple = λX. double [X→X] (double [X]);
▶ quadruple : ∀X. (X→X) → X → X
```

12.4.1.1 Exercise [Quick check]: Using the typing rules above, convince yourself that these terms have the types given. □

12.4.2 Polymorphic Lists

12.4.3 Impredicative Encodings

Here are the “Church booleans”:

```
CBool = ∀X.X→X→X;

tt = λX. λt:X. λf:X. t;
ff = λX. λt:X. λf:X. f;

cif = λb:CBool. λX. λth:X. λel:X. b [X] th el;
```

```
▶ tt : ∀X. X → X → X
  ff : ∀X. X → X → X
  cif : CBool → (∀X. X → X → X)
```

In fact, `cif` is the identity function! In other words, an element of type `CBool` is itself a “conditional,” in the sense that it takes two alternatives and chooses either the first or the second depending on whether it represents `true` or `false`.

We can write common boolean operations like `not` either in terms of `cif`

```
not = λb:CBool. cif b [CBool] ff tt;
```

```
▶ not : CBool → CBool
```

or, more interestingly, by directly constructing a new boolean from the given one:

```
not = λb:CBool.
         λX. λt:X. λf:X.
           b [X] f t;
```

```
▶ not : CBool → (∀X. X → X → X)
```

12.4.3.1 Exercise [Recommended]: Write a term

`and : CBool→CBool→CBool`

without using `cif`. □

We can play a similar game with numbers. The elements of the type

```
CNat = ∀X. (X→X) → X → X;
```

of “Church numerals” can be used to represent natural numbers as follows:

```
czero  =  λX. λs:X→X. λz:X. z;
cone   =  λX. λs:X→X. λz:X. s z;
ctwo   =  λX. λs:X→X. λz:X. s (s z);
cthree =  λX. λs:X→X. λz:X. s (s (s z));
```

```
▶ czero : ∀X. (X→X) → X → X
  cone : ∀X. (X→X) → X → X
  ctwo : ∀X. (X→X) → X → X
  cthree : ∀X. (X→X) → X → X
```

and so on. That is, a church numeral `n` is a function that, given arguments `s` and `z`, applies `s` to `z`, `n` times.

The successor function can be defined as follows:

```
csucc = λn:CNat.
          λX. λs:X→X. λz:X.
            s (n [X] s z);
▶ csucc : CNat → (∀X. (X→X) → X → X)
```

That is, `csucc n` returns an element of `CNat` that, given `s` and `z`, applies `s` to `z`, `n` times (by applying `n`), and then once more.

Other arithmetic operations can be defined similarly:

```
cplus = λm:CNat. λn:CNat.
          m [CNat] csucc n;
▶ cplus : CNat → CNat → CNat
```

Or, more directly:

```
cplus = λm:CNat. λn:CNat.
          λX. λs:X→X. λz:X.
            m [X] s (n [X] s z);
▶ cplus : CNat → CNat → (∀X. (X→X) → X → X)
```

We can convert from church numerals to ordinary numbers like this:

```
cnat2nat = λm:CNat. m [Nat] (λx:Nat. succ(x)) 0;
▶ cnat2nat : CNat → Nat
```

This allows us to verify that the terms we have defined actually compute the desired arithmetic functions:

```
cnat2nat (cplus (csucc czero) (csucc (csucc czero)));
▶ 3 : Nat
```

12.4.3.2 Exercise [Recommended]: Write a function `iszero` that returns `true` when applied to the church numeral `czero` and `false` otherwise. □

12.4.3.3 Exercise: Show that the terms

```
   ctimes = λm:CNat. λn:CNat.
              λX. λs:X→X.
                 n [X] (m [X] s);

   cexp = λm:CNat. λn:CNat.
            λX.
              n [X→X] (m [X]);
▶ ctimes : CNat → CNat → (∀X. (X→X) → X → X)
   cexp : CNat → CNat → (∀X. (X→X) → X → X)
```

have the indicated types. Sketch an informal argument that they implement the arithmetic multiplication and exponentiation operators. □

So, if we wanted to, we could actually drop `Nat` and iteration from the language. This doesn't affect real programming languages based on System F, since we want to build in booleans for efficiency and syntactic convenience, but it helps when we're doing theoretical work because it keeps the system small.

12.4.3.4 Exercise [Recommended]: In what sense does the type

```
PairNat = ∀X. (CNat→CNat→X) → X;
```

represent pairs of numbers? Write functions

```
pairNat : CNat→CNat→PairNat;
fstNat : PairNat→CNat;
sndNat : PairNat→CNat;
```

for constructing elements of this type from pairs of numbers and for accessing their first and second components. □

12.4.3.5 Exercise [Moderately difficult, recommended]: Use the functions defined in Exercise 12.4.3.4 to write a function `pred` that computes the predecessor of a church numeral (returning `0` if its input is `0`). (Hint: the key idea is developed in the example in Section **??**. Define a function `f : PairNat→PairNat` that maps the pair (i, j) into $(i + 1, i)$—that is, it throws away the second component of its argument, copies the first component to the second, and increments the first. Then n applications of `f` to the starting pair $(0, 0)$ yields the pair $(n, n - 1)$...) □

12.4.3.6 Exercise [Optional]: There is another way of computing the predecessor function on church numerals. Let `K` stand for the untyped lambda-term `λx. λy. x` and `I` for `λx. x`. The untyped lambda-term

vpred = `[n] λs. λz. n (λp. λq. q (p s)) (K z) I`

(due to J. Velmans) computes the predecessor of an untyped Church numeral. Show that this term can be typed in λF by adding type abstractions and applications as necessary and annotating the bound variables in the untyped term with appropriate types. For extra credit, explain why it works! (Solution on page **??**.) [Thanks to Michael Levin for making me aware of this example.] □

12.5 Metatheory

12.5.1 Subject Reduction

The subject reduction property from the simply typed lambda-calculus holds exactly the same here. To prove it, we need one additional substitution lemma, for types.

12.5.2 Strong Normalization

The strong normalization property for System F, proved using an extension of the method presented in Chapter **??**, was one of the major achievements of Girard's Ph.D. thesis. Since then, this proof technique has been studied and reworked by many authors, including [**?**].

12.5.2.1 Theorem [Strong normalization]: All well-typed $\lambda\mathtt{F}$ terms are strongly normalizing. □

12.5.3 Erasure and Typeability

There are two reasonable definitions of erasure for $\lambda\mathtt{F}$*: the "type-passing" and "type-forgetting" interpretations...*

12.5.3.1 Exercise: The strong normalization property of System $\mathtt{F}$ implies that the term

$$\Omega \quad = \quad \mathtt{(\lambda x.\ x\ x)\ (\lambda y.\ y\ y)}$$

in the untyped lambda-calculus cannot be typed in System $\mathtt{F}$, since reduction of Ω never reaches a normal form. However, it is possible to give a more direct, "combinatorial" proof of this fact, using just the rules defining the typing relation.

1. Let us call a System $\mathtt{F}$ term **exposed** if it is a variable, an abstraction $\mathtt{\lambda x{:}T.t}$, or an application $\mathtt{t\ s}$ (i.e., if it is *not* a type abstraction $\mathtt{\lambda X.t}$ or type application $\mathtt{t\ S}$).

   Show that if $\mathtt{t}$ is well typed (in some context) and $\mathit{erase}(\mathtt{t}) = \mathtt{M}$, then there is some exposed term $\mathtt{s}$ such that $\mathit{erase}(\mathtt{s}) = \mathtt{M}$ and $\mathtt{s}$ is well typed (possibly in a different context).

2. Write $\mathtt{\lambda\overline{X}.t}$ as shorthand for a nested sequence of type abstractions of the form $\mathtt{\lambda X_1 \ldots . \lambda X_n .t}$. Similarly, write $\mathtt{t\ [\overline{A}]}$ for a nested sequence of type applications $\mathtt{((t\ [A_1])\ldots[A_{n-1}])\ [A_n]}$ and $\mathtt{\forall\overline{X}.T}$ for a nested sequence of polymorphic types $\mathtt{\forall X_1 \ldots . \forall X_n .T}$. Note that these sequences are allowed to be empty. For example, if $\mathtt{\overline{X}}$ is the empty sequence of type variables, then $\mathtt{\forall\overline{X}.T}$ is just $\mathtt{T}$.

Show that if *erase*($\mathtt{t}$) = $\mathtt{M}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}$, then there exists some $\mathtt{s}$ of the form $\lambda\overline{\mathtt{X}}.\ (\mathtt{u}\ [\overline{\mathtt{A}}])$, for some sequence of type variables $\overline{\mathtt{X}}$, some sequence of types $\overline{\mathtt{A}}$, and some exposed term $\mathtt{u}$, with *erase*($\mathtt{s}$) = $\mathtt{M}$ and $\Gamma \vdash \mathtt{s} : \mathtt{T}$.

3. Show that if $\mathtt{t}$ is an exposed term of type $\mathtt{T}$ (under Γ) and *erase*($\mathtt{t}$) = $\mathtt{M\ N}$, then $\mathtt{t}$ has the form $\mathtt{s\ u}$ for some terms $\mathtt{s}$ and $\mathtt{u}$ such that *erase*($\mathtt{s}$) = $\mathtt{M}$ and *erase*($\mathtt{u}$) = $\mathtt{N}$, with $\Gamma \vdash \mathtt{s} : \mathtt{U}{\to}\mathtt{T}$ and $\Gamma \vdash \mathtt{u} : \mathtt{U}$.

4. Suppose that $\Gamma(\mathtt{x}) = \mathtt{T}$. Show that if $\Gamma \vdash \mathtt{u} : \mathtt{U}$ and *erase*($\mathtt{u}$) = $\mathtt{x\ x}$, then either

   (a) $\mathtt{T} = \forall\overline{\mathtt{X}}.\mathtt{X}_i$, where $\mathtt{X}_i \in \overline{\mathtt{X}}$, or else

   (b) $\mathtt{T} = \forall\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2.\mathtt{T}_1{\to}\mathtt{T}_2$, where $\{\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2 \mapsto \overline{\mathtt{A}}\}\mathtt{T}_1 = \{\overline{\mathtt{X}}_1 \mapsto \overline{\mathtt{B}}\}(\forall\overline{\mathtt{Z}}.\mathtt{T}_1{\to}\mathtt{T}_2)$ for some sequences of types $\overline{\mathtt{A}}$ and $\overline{\mathtt{B}}$ with $|\overline{\mathtt{A}}| = |\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2|$ and $|\overline{\mathtt{B}}| = |\overline{\mathtt{X}}_1|$.

5. Show that if *erase*($\mathtt{s}$) = $\lambda\mathtt{x}.\mathtt{M}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\mathtt{S}$ has the form $\forall\overline{\mathtt{X}}.\mathtt{S}_1{\to}\mathtt{S}_2$, for some $\overline{\mathtt{X}}$, $\mathtt{S}_1$, and $\mathtt{S}_2$.

6. Define the **leftmost leaf** of a type $\mathtt{T}$ as follows:

$$
\begin{array}{lcl}
\textit{leftmost-leaf}(\mathtt{X}) & = & \mathtt{X} \\
\textit{leftmost-leaf}(\mathtt{S}{\to}\mathtt{T}) & = & \textit{leftmost-leaf}(\mathtt{S}) \\
\textit{leftmost-leaf}(\forall\mathtt{X}.\mathtt{S}) & = & \textit{leftmost-leaf}(\mathtt{S}).
\end{array}
$$

   Show that if $\{\overline{\mathtt{X}}_1\overline{\mathtt{X}}_2 \mapsto \overline{\mathtt{A}}\}(\forall\overline{\mathtt{Y}}.\mathtt{T}_1) = \{\overline{\mathtt{X}}_1 \mapsto \overline{\mathtt{B}}\}(\forall\overline{\mathtt{Z}}.(\forall\overline{\mathtt{Y}}.\mathtt{T}_1){\to}\mathtt{T}_2)$, then *leftmost-leaf*($\mathtt{T}_1$) = $\mathtt{X}_i$ for some $\mathtt{X}_i \in \overline{\mathtt{X}}_1\overline{\mathtt{X}}_2$.

7. Show that Ω is not typeable in System $\mathtt{F}$.

(Solution on page **??**.) □

The **type erasure** function for λF is the following mapping from λF to terms in the untyped lambda-calculus:

$$
\begin{array}{lcl}
\textit{erase}(\mathtt{x}) & = & \mathtt{x} \\
\textit{erase}(\lambda\mathtt{x}{:}\mathtt{T}.\ \mathtt{t}) & = & \lambda\mathtt{x}.\ \textit{erase}(\mathtt{t}) \\
\textit{erase}(\mathtt{t\ s}) & = & \textit{erase}(\mathtt{t})\ \textit{erase}(\mathtt{s}) \\
\textit{erase}(\lambda\mathtt{X}.\ \mathtt{t}) & = & \textit{erase}(\mathtt{t}) \\
\textit{erase}(\mathtt{t}\ [\mathtt{S}]) & = & \textit{erase}(\mathtt{t})
\end{array}
$$

A term $\mathtt{M}$ in the untyped lambda-calculus is said to be **typeable** in λF if there is some well-typed term $\mathtt{t}$ such that *erase*($\mathtt{t}$) = $\mathtt{M}$.

12.5.4 Type Reconstruction

12.6 Implementation

12.6.1 Nameless Representation of Types

12.6.2 ML Code

```
type ty =
    ...
  | TyVar of int * int
  | TyAll of string * ty

type term =
    ...
  | TmTAbs of info * string * term
  | TmTApp of info * term * ty

let tyshifti d c tyT =
  tymap
    (fun c x n → if x>=c then
                      if x+d<0 then err "Scoping error!"
                      else TyVar(x+d,n+d)
                    else TyVar(x,n+d))
    c
    tyT

let tmshifti d c t =
  tmmap
    (fun fi c x n → if x>=c then TmVar(fi,x+d,n+d) else TmVar(fi,x,n+d))
    (tyshifti d)
    c
    t

let tmshift t d = tmshifti d 0 t

let tyshift tyT d = tyshifti d 0 tyT
```

(Note that we also need to shift the type annotation in the `TmAbs` case.)

```
let tmsubsti s j t =
  tmmap
    (fun fi j x n → if x=j then (tmshift s j) else TmVar(fi,x,n))
    (fun j tyT → tyT)
    j
    t
```

```
let tmsubst s t = tmsubsti s 0 t

let tmsubstsnip s t = tmshift (tmsubst (tmshift s 1) t) (-1)

let tysubsti tyS j tyT =
  tymap
    (fun j x n → if x=j then (tyshift tyS j) else (TyVar(x,n)))
    j
    tyT

let tysubst tyS tyT = tysubsti tyS 0 tyT

let tysubstsnip tyS tyT = tyshift (tysubst (tyshift tyS 1) tyT) (-1)

let rec tytmsubsti tyS j t =
  tmmap
    (fun fi c x n → TmVar(fi,x,n))
    (fun j tyT → tysubsti tyS j tyT)
    j
    t

let tytmsubst tyS t = tytmsubsti tyS 0 t

let tytmsubstsnip tyS t = tmshift (tytmsubst (tyshift tyS 1) t) (-1)
```

12.7 Further Reading

Chapter 13

Existential Types

Having seen the role of universal quantifiers in a type system, one might wonder whether *existential* quantifiers would also be useful in programming. In fact, they provide an elegant foundation for data abstraction and "information hiding," as we shall see in this chapter.

13.1 Motivation

The polymorphic types in Chapter 12 can be viewed in two different ways:

1. A *logical intuition* is that an element of the type $\forall$X.T is a value that has type $\{X \mapsto S\}T$ for any choice of S.

   This intuition corresponds to a "type-erasure view" of what a term means: for example, the polymorphic identify function λX.λx:X.x erases to the untyped identity function λx.x, which maps an argument of any type S to a result of the same type S.

2. A more *operational intuition* is that an element of $\forall$X.T is a function mapping a type S to a concrete instance with type $\{X \mapsto S\}T$.

   This intuition corresponds to our definition of λF in Chapter 12, where the reduction of a type application is considered an actual step of the computation.

Similarly, there are two different ways of looking at an existential type {$\exists$X,T}:

1. The *logical intuition* is that an element of {$\exists$X,T} has type $\{X \mapsto S\}T$ for *some* type S.

2. The *operational intuition* is that an element of {$\exists$X,T} is a pair of a type S and a term t of type $\{X \mapsto S\}T$.

We will take an operational view of existentials in this chapter, since it provides a closer analogy between existential packages and modules or abstract data types. Our concrete syntax for existential types (`{∃X,T}` rather than the more standard `∃X.T`) reflects this analogy.

As always, to understand existential types we need to know two things: how to *build* (or "introduce," in the jargon of Section **??**) elements of existential types, and how to *use* (or "eliminate") these values in computations.

A value `{∃X=S,t}` of type `{∃X,T}` can be thought of as a simple module with one (hidden) type component and one term component.[1] The type `S` is often called the **hidden representation type**, or sometimes (to emphasize the connection with intuitionistic logic) the **witness type** of the package. As a simple example, the package

```
p = {∃X=Nat, {a=5, f=λx:Nat. succ(x)}}
```

has the existential type

```
{∃X, {a:X, f:X→X}}.
```

That is, the right-hand component of `p` is a record containing a field `a` of type `X` and a field `f` of type `X→X`, for some `X` (namely `Nat`).

The same package `p` *also* has the type

```
{∃X, {a:X, f:X→Nat}},
```

since its right-hand component is a record with fields `a` and `f` of type `X` and `X→Nat`, for some `X` (namely `Nat`). This example shows that, in general, we can't make an automatic decision about which existential type a given package belongs to: the programmer must specify which one is intended. The simplest way to do this is just to add an annotation to every package that explicitly gives its intended type. So we'll write

```
p1 = {∃X=Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▸ p1 : {∃X, {a:X,f:X→X}}
```

or:

```
p2 = {∃X=Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▸ p2 : {∃X, {a:X,f:X→Nat}}
```

[1]Obviously, we could generalize to many type/term components, but we'll stick with just one of each to keep the notation tractable. The effect of multiple type components can be achieved by nesting single-type existentials, while the effect of multiple term components can be achieved by using a tuple or record as the right-hand component:

$$\{\exists X_1{=}S_1,\ \exists X_2{=}S_2,\ t_1,\ t_2\} \stackrel{\text{def}}{=} \{\exists X_1{=}S_1,\ \{\exists X_2{=}S_2,\ \{t_1,\ t_2\}\}\}$$

The type annotation introduced by `as` is similar to the coercion construct introduced in Section **??**, which allows *any* term to be annotated with its intended type. We are essentially requiring a single coercion as part of the concrete syntax of the package construct.

The complete typing rule for packages is as follows:

$$\frac{\Gamma \vdash \texttt{t}_2 : \{\texttt{X} \mapsto \texttt{T}_1\}\texttt{T}_2}{\Gamma \vdash \texttt{\{}\exists\texttt{X=T}_1\texttt{,t}_2\texttt{\} as \{}\exists\texttt{X,T}_2\texttt{\}} : \texttt{\{}\exists\texttt{X,T}_2\texttt{\}}} \qquad \text{(T-PACK)}$$

Of course, packages with different representation types can inhabit the same existential type. For example:

```
p3 = {∃X=Nat, 0} as {∃X,X};
p4 = {∃X=Bool, true} as {∃X,X};
```

```
▸ p3 : {∃X, X}
  p4 : {∃X, X}
```

Or, more usefully:

```
p5 = {∃X=Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
p6 = {∃X=Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}};
```

```
▸ p5 : {∃X, {a:X,f:X→Nat}}
  p6 : {∃X, {a:X,f:X→Nat}}
```

13.1.1 Exercise [Quick check]: Here are three more variations on the same theme:

```
p7 = {∃X=Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
p8 = {∃X=Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat→X}};
p9 = {∃X=Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:Nat, f:Nat→Nat}};
```

```
▸ p7 : {∃X, {a:X,f:X→X}}
  p8 : {∃X, {a:X,f:Nat→X}}
  p9 : {∃X, {a:Nat,f:Nat→Nat}}
```

In what ways are these less useful than `p5` and `p6`? (Solution on page **??**.) □

A useful intuition for the existential elimination construct comes from the analogy with modules. If an existential package is a simple form of module, then package elimination is like an `open` or `import` construct: it allows the components of the module to be used in some other part of the program, but holds the identity of the module's type component abstract. This can be achieved with a kind of pattern-matching binding:

$$\frac{\Gamma \vdash \texttt{t}_1 : \exists\texttt{X.T}_{11} \qquad \Gamma, \texttt{X}, \texttt{x:T}_{11} \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let \{X,x\}=t}_1 \texttt{ in t}_2 : \texttt{T}_2} \qquad \text{(T-UNPACK)}$$

That is, if $\texttt{t}_1$ is an expression that yields an existential package, then we can bind its type and term components to the pattern variables `X` and `x` and use them in computing $\texttt{t}_2$.

For example, suppose that `p` has the following existential type:

```
▸ p : {∃X, {a:X,f:X→Nat}}
```

Then the elimination expression

```
    let {X,x}=p in (x.f x.a);
▸ 1 : Nat
```

opens p and uses its components (x.f and x.a) to compute a numeric result. The body of the elimination form is also permitted to use the type variable X, as in the following example:

```
    let {X,x}=p in (λy:X. x.f y) x.a;
▸ 1 : Nat
```

The fact that the package's representation type is held abstract during the typechecking of the body (t_2) means that the only operations allowed on x are those warranted by its "abstract type" T_1. In the present example, we are not allowed to use x.a concretely as a number:

```
    let {X,x}=p in succ(x.a);
▸ Error: argument of succ is not a number
```

This restriction makes good sense, since we saw above that a package p with the given existential type might use either Nat or Bool as its representation type.

There is another, more subtle, way in which typechecking of the existential elimination construct may fail. In the rule T-UNPACK, the type variable X appears in the context in which t_2's type is calculated, but does *not* appear in the context of the rule's conclusion. This means that the result type T_2 cannot contain X free, since any free occurrences of X will be out of scope in the conclusion. More operationally, in terms of the nameless presentation of terms discussed in Section **??**, the T-UNPACK rule proceeds in three steps:

1. Check the subexpression t_1 and ensure that it has an existential type $\{\exists X.T_{11}\}$.
2. Extend the context Γ with X and $x:T_{11}$ and check that t_2 has some type T_2.
3. Shift the indices of free variables in T_2 *down* by two, so that it makes sense with respect to Γ.
4. Return the resulting type as the type of the whole let...in... expression.

Clearly, if X occurs free in T_2, then the shifting step will yield a nonsensical type containing free variables with negative indices; typechecking must fail at this point.

```
    let {X,x}=p in x.a;
▸ Error: Scoping error!
```

The computation rule for existentials is straightforward: if the package subexpression has already been reduced to a concrete package, then we may substitute the components of the package for the variables `X` and `x` in the body $\texttt{t}_2$:

In terms of the analogy with modules, this rule can be viewed as a kind of "linking" operation, in which symbolic names (`X` and `x`) referring to the components of a separately compiled module are replaced by direct references to the actual contents of the module.

Since the type variable `X` is substituted away by this rule, the resulting program actually has concrete access to the package's internals. This is just another example of a phenomenon we have seen several times: expressions can become "more typed" as computation proceeds, and in particular an ill-typed expression can reduce to a well-typed one.

□**:** **Extends**

New syntactic forms:

`t`	::=	...	
		`{∃X=T,t} as T`	*packing*
		`let {X,x}=t in t`	*unpacking*
`v`	::=	...	
		`{∃X=T,v} as T`	*package value*
`T`	::=	...	
		`{∃X,T}`	*existential type*

New evaluation rules:

$$\texttt{let \{X,x\}=\{}\exists\texttt{X=T}_{11}\texttt{,v}_{12}\texttt{\} in t}_2 \longrightarrow \{\texttt{X} \mapsto \texttt{T}_{11}\}\{\texttt{x} \mapsto \texttt{v}_{12}\}\texttt{t}_{12} \qquad \text{(EV-PACKBETA)}$$

$$\frac{\texttt{t}_{12} \longrightarrow \texttt{t}'_{12}}{\texttt{\{}\exists\texttt{X=T}_{11}\texttt{,t}_{12}\texttt{\} in t}_2 \longrightarrow \texttt{\{}\exists\texttt{X=T}_{11}\texttt{,t}'_{12}\texttt{\} in t}_2} \qquad \text{(EV-PACK)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}'_1}{\texttt{let \{X,x\}=t}_1\texttt{ in t}_2 \longrightarrow \texttt{let \{X,x\}=t}_1\texttt{ in t}_2} \qquad \text{(EV-UNPACK)}$$

New typing rules:

$$\frac{\Gamma \vdash \mathtt{t}_2 : \{\mathtt{X} \mapsto \mathtt{T}_1\}\mathtt{T}_2}{\Gamma \vdash \mathtt{\{\exists X=T_1,t_2\}\ as\ \{\exists X,T_2\}} : \mathtt{\{\exists X,T_2\}}} \qquad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists\mathtt{X}.\mathtt{T}_{11} \qquad \Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T}_{11} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ \{X,x\}=t_1\ in\ t_2} : \mathtt{T}_2} \qquad \text{(T-UNPACK)}$$

13.2 Data Abstraction with Existentials

13.2.1 Abstract Data Types

For a more interesting example, here is a simple package defining an **abstract data type** of (purely functional) counters.

```
counterADT =
   {∃Counter = Nat,
    {new = 0,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};
▸ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

The concrete representation of a counter is just a number. The package provides three operations on counters: a constant `new`, a function `get` for extracting a counter's current value, and a function `inc` for creating a new counter whose stored value is one more than the given counter's. Having created the counter package, we next open it, exposing the operations as the fields of a record `counter`:

```
let {Counter,counter}=counterADT in
counter.get (counter.inc counter.new);
▸ 1 : Nat
```

If we organize our code so that the body of this `let` contains the whole remainder of the program, then this idiom

```
let {Counter,counter} = <counter package> in
<rest of program>
```

has the effect of declaring a fresh type `Counter` and a variable `counter` of type `{new:Counter,get:Counter→Nat,inc: Counter→Counter}`.

It is instructive to compare the above with a more standard abstract data type declaration, such as might be found in a program in Ada [oD80] or Clu [LAB+81]:

```
ADT counter =
  type Counter
  representation Nat
  operations
    {new = 0
         : Counter,
     get = λi:Nat. i
         : Counter→Nat,
     inc = λi:Nat. succ(i)}
         : Counter→Counter};

counter.get (counter.inc counter.new);
```

The version using existential types is somewhat harder to read, compared to the syntactically sugared second version, but otherwise the two programs are essentially identical.

Note that we can substitute an alternative implementation of the `Counter` ADT—for example, one where the internal representation is a record containing a `Nat` rather than just a single `Nat`

```
counterADT =
   {∃Counter = {x:Nat},
    {new = {x=0},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};

▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

in complete confidence that the whole program will remain typesafe, since we are guaranteed that the rest of the program cannot access instances of `Counter` except using `get` and `inc`. This is the essence of data abstraction by information hiding.

In the body of the program, the type name `Counter` can be used just like the base types built into the language. We can define functions that operate on counters:

```
let {Counter,counter}=counterADT
in

let addthree = λc:Counter.
                 counter.inc (counter.inc (counter.inc c))
in

counter.get (addthree counter.new);
```

```
▶ 3 : Nat
```

We can even define new abstract data types whose representation involves counters. For example, the following program defines an ADT of flip-flops, using a counter as the (not very efficient) representation type:

```
let {Counter,counter} =
     {∃Counter = Nat,
      {new = 0,
       get = λi:Nat. i,
       inc = λi:Nat. succ(i)}}
   as {∃Counter,
       {new: Counter,
        get: Counter→Nat,
        inc: Counter→Counter}}
in

let {FlipFlop,flipflop} =
     {∃FlipFlop = Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop,
        read:   FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop,
        reset:  FlipFlop→FlipFlop}}
in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
```

```
▶ false : Bool
```

13.2.1.1 Exercise [Recommended]: Follow the model of the above example to define an abstract data type of *stacks* of numbers, with operations `new`, `push`, `pop`, and `isempty`. Use the `List` type introduced in Exercise **??** as the underlying representation. Write a simple main program that creates a stack, pushes a couple of numbers onto it, pops off the top element, and returns it.

This exercise is best done on-line. Use the checker named "`everything`" and copy the contents of the file `test.f` from the `everything` directory (which contains definitions of the `List` constructor and associated operations) to the top of your own input file. □

13.2.2 Existential Objects

The sequence of "pack then open" that we saw in the last section is the hallmark of ADT-style programming using existential packages. A package defines an abstract type and its associated operations, and each package is opened immediately after it is built, binding a type variable for the abstract type and exposing the ADT's operations abstractly, with this variable in place of the concrete representation type. Existential types can also be used to model other common types of data abstraction. In this section, we show how a simple form of objects can be understood in terms of a different idiom based on existentials.

We will again use simple counters as our running example, as we did both in the previous section on existential ADTs and in our previous encounter with objects, in Chapter 11. Unlike the counters of Chapter 11, however, the counter objects in this section will be purely functional: sending the message `inc` to a counter will not change its internal state in-place, but rather will return a *fresh* counter object with incremented internal state.

A counter object, then, will comprise two basic components: a number (its internal state), and a pair of methods, `get` and `inc`, that can be used to query and update the state. We also need to ensure that the only way that the state of a counter object can be queried or updated is by using one of its two methods. This can be accomplished by wrapping the state and methods in an existential package, abstracting the type of the internal state. For example, a counter object holding the value 5 might be written

```
c = {∃X = Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
 as Counter;
```

where:

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

To use a method of a counter object, we will need to open it up and apply the appropriate element of its `methods` to its `state` field. For example, to get the current value of `c` we can write:

```
  let {X,body} = c in
          body.methods.get(body.state);
▸ 5 : Nat
```

More generally, we can define a little function that "sends the `get` message" to any counter:

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
```

```
▶ sendget : Counter → Nat
```

Invoking the `inc` method of a counter object is a little more complicated. If we simply do the same as for `get`, the typechecker complains

```
 let {X,body} = c in
          body.methods.inc(body.state);
▶ Error: Scoping error!
```

because the type variable `X` appears free in the type of the body of the `let`. Indeed, what we've written doesn't make intuitive sense either, since the result of the `inc` method is a "bare" internal state, not an object. To satisfy both the typechecker and our informal understanding of what invoking `inc` should do, we must take this fresh internal state and "repackage" it as a counter object, using the same record of methods and the same internal state type as in the original object:

```
c1 = let {X,body} = c in
       {∃X = X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
 as Counter;
```

More generally, to "send the `inc` message" to an arbitrary counter object, we can write:

```
sendinc = λc:Counter.
            let {X,body} = c in
              {∃X = X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter
```

More complex operations on counters can be implemented in terms of these two basic operations:

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▶ addthree : Counter → Counter
```

13.2.2.1 Exercise: Implement `FlipFlop` objects with `Counter` objects as their internal representation type, following the model of the `FlipFlop` ADT in Section 13.2.1. □

13.2.3 Objects vs. ADTs

What we have seen in Section 13.2.2 falls significantly short of a full-blown model of object-oriented programming. Many of the features that we saw in Chapter 11, including subtyping, classes, inheritance, and recursion through `self` and `super`, are missing here. We will come back to modeling these features in later chapters, when we have added a few necessary refinements to our modeling language. But even for the simple objects we have developed so far, there are several interesting comparisons to be made with ADTs.

At the coarsest level, the two programming idioms fall at opposite ends of a spectrum: when programming with ADTs, packages are opened immediately after they are built; on the other hand, when packages are used to model objects they are kept closed as long as possible—until the moment when they *must* be opened so that one of the methods can be applied to the internal state.

A consequence of this difference is that the "abstract type" of counters refers to different things in the two styles. In an ADT-style program, the counter values manipulated by client code such as `addthree` are elements of the underlying representation type (e.g., simple numbers). In an object-style program, a counter value is a whole package—not only a number, but also the implementations of the `get` and `inc` methods. This stylistic difference is reflected in the fact that, in the ADT style, the type `Counter` is a bound type variable introduced by the `let` construct, while in the object style `Counter` abbreviates the whole existential type `{∃X, {state:X, methods: {get:X→Nat,inc:X→X}}}`. Thus:

- All the counter values generated from the counter ADT are elements of the same internal representation type; there is a single implementation of the counter operations that works on this internal representation.
- Each counter object, on the other hand, carries its own representation type and its own set of methods that work for this representation type.

13.2.3.1 Exercise: In what ways do the *classes* found in mainstream object-oriented languages like C++ and Java resemble the simple object types discussed here? In what ways do they resemble ADTs? □

13.3 Encoding Existentials

The encoding of pairs as a polymorphic type in Exercise 12.4.3.4 suggests a similar encoding for existential types in terms of universal types, using the intuition that an element of an existential type is a pair of a type and a value:

$$\{\exists X,T\} \overset{\text{def}}{=} \forall Y.\ (\forall X.\ T \to Y) \to Y.$$

That is, an existential package is thought of as a data value that, given a result type and a "continuation," calls the continuation to yield a final result. The continuation takes two arguments—a type `X` and a value of type `T`—and uses them in computing the final result.

Given this encoding of existential types, the encoding of the packaging and unpackaging constructs is essentially forced. To encode a package

```
{∃X=S,t} as {∃X,T}
```

we must exhibit a value of type `∀Y. (∀X. T→Y) → Y`. This type begins with a universal quantifier, the body of which is an arrow. An element of this type should therefore begin with two abstractions:

```
{∃X=S,lt} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). ...
```

To complete the job, we need to return a result of type `Y`; clearly, the only way to do this is to apply `f` to some appropriate arguments. First, we supply the type `S` (this is a natural choice, being the only type we have lying around at the moment):

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] ...
```

Now, the type application `f [S]` has type `{X ↦ S}(T→Y)`, i.e., `({X ↦ S}T)→Y`. We can thus supply `t` (which, by rule T-PACK, has type `{X ↦ S}T`) as the next argument:

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] t
```

The type of the whole application `f [S] t` is now `Y`, as required.

To encode the unpacking construct

$$\texttt{let \{X,x\}=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{...}$$

we proceed as follows. First, the typing rule T-UNPACK tells us that $\texttt{t}_1$ should have some type $\texttt{\{∃X,T}_{11}\texttt{\}}$, that $\texttt{t}_2$ should have type $\texttt{T}_2$ (under an extended context binding `X` and $\texttt{x:T}_{11}$), and that $\texttt{T}_2$ is the type we expect for the whole `let...in...` expression.[2] As in the Church encodings in Section 12.4.3, the intuition here is that the introduction form (`{∃X=S,t}`) is encoded as an active value that "performs its own elimination." So the encoding of the elimination form here should simply take the existential package $\texttt{t}_1$ and apply it to enough arguments to yield a result of the desired type $\texttt{T}_2$:

$$\texttt{let \{X,x\}=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{t}_1\texttt{ ...}$$

The first argument to $\texttt{t}_1$ should be the desired result of the whole expression, i.e., $\texttt{T}_2$:

$$\texttt{let \{X,x\}=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{t}_1\texttt{ [T}_2\texttt{] ...}$$

[2] Strictly speaking, the fact that the translation requires these extra bits of type information not present in the syntax of terms means that what we are translating is actually *typing derivations*, not terms.

Now, the application $\mathtt{t_1\ [T_2]}$ has type $(\forall\mathtt{X.\ T}\to\mathtt{T_2}) \to \mathtt{T_2}$. That is, if we can now supply another argument of type $(\forall\mathtt{X.T}\to\mathtt{T_2})$, we will be finished. Such an argument can be obtained by *abstracting* the body $\mathtt{t_2}$ on the variables X and x:

$$\mathtt{let\ \{X,x\}=t_1\ in\ t_2} \quad \stackrel{\text{def}}{=} \quad \mathtt{t_1\ [T_2]\ (\lambda X.\ \lambda x{:}T_{11}.\ t_2).}$$

This finishes the encoding.

13.3.1 Exercise: What must we prove to show that our encoding of existentials is correct? □

13.3.2 Exercise [Recommended]: Take a blank piece of paper and, without looking at the above encoding, regenerate it from scratch. □

13.3.3 Exercise: Can universal types be encoded in terms of existential types? □

13.4 Implementation

```
type ty =
    ...
  | TySome of string * ty

type term =
    ...
  | TmPack of info * string * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

13.5 Historical Notes

The correspondence between ADTs and existential types was first developed by Mitchell and Plotkin [MP88]. (They also noticed the correspondance with objects.)

Chapter 14

Bounded Quantification

Many of the interesting problems in type systems arise from the *combination* of features that, in themselves, may be quite simple. In this chapter, we encounter our first substantial example: a system that mixes subtyping with polymorphism.

The most basic combination of these features is actually quite straightforward. We simply add to the subtyping relation a rule for comparing quantified types:

$$\frac{\mathtt{S} <: \mathtt{T}}{\forall \mathtt{X}.\mathtt{S} <: \forall \mathtt{X}.\mathtt{T}}$$

In this chapter, we consider a more interesting combination, in which the syntax, typing, and subtyping rules for universal quantifiers are actually refined to take subtyping into account. The resulting notion of **bounded quantification** substantially increases both the expressive power of the system and its metatheoretic complexity.

14.1 Motivation

To see why we might want to combine subtyping and polymorphism in this more intimate manner, consider the identity function on records with a numeric field `a`:

```
  f = λx:{a:Nat}. x;
▶ f : {a:Nat} → {a:Nat}
```

If we define a record of this form

```
  ra = {a=0};
```

then we can apply `f` to `ra` (in any of the type systems that we have seen), yielding a record of the same form.

```
  (f ra);
```

```
▶ {a=0} : {a:Nat}
```

If we define a larger record `rab` with two fields, `a` and `b`,

```
rab = {a=0, b=true};
```

we can also apply `f` to `rab`, using the rule of subsumption introduced in Chapter 10.

```
(f rab);
▶ {a=0, b=true} : {a:Nat}
```

However, the type of the result has only the field `a`, which means that a term like `(f rab).b` will be judged ill typed. In other words, by passing `rab` through the identity function, we have lost the ability to access its `b` field!

Using the polymorphism of System F, we can write `f` in a different way:

```
fpoly = λX. λx:X. x;
▶ fpoly : ∀X. X → X
```

The application of `fpoly` to `rab` (and an appropriate type argument) yields the desired result:

```
(fpoly [{a:Nat, b:Bool}] rab);
▶ {a=0, b=true} : {a:Nat, b:Bool}
```

But in making the type of `x` into a variable, we have given up some information that `f` might have wanted to use. For example, suppose we intend that `f` return a pair of its original argument and the numeric successor of its `a` field.

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▶ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

Again, we can apply `f2` to both `ra` and `rab`, losing the `b` field in the second case.

```
(f2 ra);
▶ {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}

(f2 rab);
▶ {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

But this time polymorphism offers us no solution. If we replace the type of `x` by a variable `X` as before, we lose the constraint that `x` must have an `a` field, which is required to compute the `asucc` field of the result.

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▶ Error: Expected record type
```

The fact about the operational behavior of `f2` that we want to express in its type is:

> `f2` takes an argument of any record type `R` that includes a numeric `a` field and returns as its result a record containing a field of type `R` and a field of type `Nat`.

We can use the subtype relation to express this concisely as follows:

> `f2` takes an argument of any subtype `R` of the type `{a:Nat}` and returns a record containing a field of type `R` and a field of type `Nat`.

This intuition can be formalized by introducing a **subtyping constraint** on the bound variable `X` of `f2poly`.

```
f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▶ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

This interaction of subtyping and polymorphism, called **bounded quantification**, leads us to a type system commonly called System $F_{<:}$ ("F sub"), which is the topic of this chapter.

14.2 Definitions

We form System $F_{<:}$ by combining the types and terms of System F with the subtype relation from Chapter 10 and refining universal quantifiers with subtyping constraints on their bound variables. When we define the subtyping rule for these bounded quantifiers, there will actually be two choices: a more tractable but less flexible rule called the **kernel** rule and a more expressive **full** subtyping rule, which will turn out to raise some unexpected difficulties when we come to designing typechecking algorithms.

Since type variables now have associated bounds (just as ordinary variables have associated types), we must keep track of them during both subtyping and typechecking. We change the type bindings in contexts to include an upper bound for each type variable, and add contexts to all the rules in the subtype relation. These bounds will be used during subtyping to justify steps of the form "the type variable `X` is a subtype of the type `T` because we assumed it was."

$$\Gamma \vdash \mathtt{X} <: \Gamma(\mathtt{X}) \qquad \text{(S-TVAR)}$$

14.2.1 Exercise [Quick check]: Exhibit a subtyping derivation showing that

$$\mathtt{B}<:\mathtt{Top},\ \mathtt{X}<:\mathtt{B},\ \mathtt{Y}<:\mathtt{X} \vdash \mathtt{B}\to\mathtt{Y} <: \mathtt{X}\to\mathtt{B}.$$ □

14.2.1 Kernel $F_{<:}$

Next, we introduce bounded universal types, extending the syntax and typing rules for ordinary universal types in the obvious way. The only rule where the extension is not completely obvious is the subtyping rule for quantified types, S-ALL. We give here the simpler variant, called the **kernel** subtyping rule for universal quantifiers, in which the bounds of the two quantifiers being compared must be identical. (The term "kernel" comes from Cardelli and Wegner's original paper [CW85], where this variant of $F_{<:}$ was called **Kernel Fun**.)

$$\Gamma \vdash X <: \Gamma(X) \qquad \text{(S-TVAR)}$$

For easy reference, here is the complete definition of kernel $F_{<:}$, with differences from previous systems highlighted:

□: **Extends**

Syntax:

t	::=	x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		λX<:T.t	*type abstraction*
		t [T]	*type application*
v	::=	x	*variable*
		λx:T.t	*abstraction value*
		λX<:U.t	*type abstraction value*
T	::=	Top	*maximum type*
		T→T	*type of functions*
		X	*type variable*
		∀X<:T.T	*universal type*
Γ	::=	∅	*empty context*
		Γ, x:T	*term variable binding*
		Γ, X<:T	*type variable binding*

Evaluation:

$$(\lambda x{:}T_2.t_1)\ v_2 \longrightarrow \{x \mapsto v_2\}t_1 \qquad \text{(EV-BETA)}$$

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(EV-APP1)}$$

$$\frac{t_2 \longrightarrow t_2{}'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(EV-APP2)}$$

$$(\lambda X{<:}U.t_1)\ T_2 \longrightarrow \{X \mapsto T_2\}t_1 \qquad \text{(EV-BETA2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ T_2 \longrightarrow t_1'\ T_2} \qquad \text{(EV-TAPP)}$$

Subtyping:

$$\Gamma \vdash \mathtt{S} <: \mathtt{S} \qquad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{U} \qquad \Gamma \vdash \mathtt{U} <: \mathtt{T}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \qquad \text{(S-TRANS)}$$

$$\Gamma \vdash \mathtt{S} <: \mathtt{Top} \qquad \text{(S-TOP)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(S-ARROW)}$$

$$\Gamma \vdash \mathtt{X} <: \Gamma(\mathtt{X}) \qquad \text{(S-TVAR)}$$

$$\frac{\Gamma, \mathtt{X} <: \mathtt{U} \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \forall \mathtt{X} <: \mathtt{U}.\mathtt{S}_2 <: \forall \mathtt{X} <: \mathtt{U}.\mathtt{T}_2} \qquad \text{(S-ALL)}$$

Typing:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{s} : \mathtt{T}} \qquad \text{(T-SUB)}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, \mathtt{X} <: \mathtt{U} \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \lambda \mathtt{X} <: \mathtt{U}.\mathtt{t}_1 : \forall \mathtt{X} <: \mathtt{U}.\mathtt{T}_1} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X} <: \mathtt{U}.\mathtt{T}_1 \qquad \Gamma \vdash \mathtt{T}_2 <: \mathtt{U}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_1} \qquad \text{(T-TAPP)}$$

14.2.2 Full $F_{<:}$

In kernel $F_{<:}$, two quantified types can only be compared if their upper bounds are identical. If we think of quantifiers as a kind of arrow types (since they classify functions from types to terms), then the kernel rule corresponds to a "covariant" version of the subtyping rule for arrows, in which the domain of an arrow type is not allowed to vary in subtypes:

$$\frac{\mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{U} \to \mathtt{S}_2 <: \mathtt{U} \to \mathtt{T}_2}$$

This restriction feels rather unnatural, both for arrows and for quantifiers. Carrying the analogy a little further, we can allow the "left-hand side" of bounded quantifiers to vary (contravariantly) during subtyping:

□ : **Extends**

New subtyping rules:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X} <: \mathtt{T_1} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \forall \mathtt{X} <: \mathtt{S_1} . \mathtt{S_2} <: \forall \mathtt{X} <: \mathtt{T_1} . \mathtt{T_2}} \qquad \text{(S-ALL)}$$

Intuitively, the "full $F_{<:}$" quantifier subtyping rule can be understood as follows. A type $\mathtt{T} = \forall \mathtt{X} <: \mathtt{T_1} . \mathtt{T_2}$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of $\mathtt{T_1}$ to instances of $\mathtt{T_2}$. If $\mathtt{T_1}$ is a subtype of $\mathtt{S_1}$, then the domain of `T` is smaller than that of $\mathtt{S} = \forall \mathtt{X} <: \mathtt{S_1} . \mathtt{S_2}$, so `S` is a stronger constraint and describes a smaller collection of polymorphic values. Moreover, if, for each type `U` that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $\mathtt{U} <: \mathtt{T_1}$), the `U`-instance of $\mathtt{S_2}$ is a subtype of the `U`-instance of $\mathtt{T_2}$, then `S` is a "pointwise stronger" constraint and again describes a smaller collection of polymorphic values.

The system with just the kernel subtyping rule for quantified types is called Kernel $F_{<:}$ (or $F^k_{<:}$). The same system with the full quantifier subtyping rule is called Full $F_{<:}$ (or $F^f_{<:}$). The bare name $F_{<:}$ refers ambiguously to both systems.

14.2.2.1 Exercise [Quick check]: Give a couple of examples of pairs of types that are related by the subtype relation of full $F_{<:}$ but are not subtypes in kernel $F_{<:}$. □

14.2.2.2 Exercise [Challenging]: Can you find any *useful* examples with this property? □

14.3 Examples

We now present some simple examples of programming in $F_{<:}$. More sophisticated uses of bounded quantification will appear in later chapters.

14.3.1 Encoding Products

In Exercise **??**, we gave the following encoding of pairs in System F. The elements of the type

```
Pair T₁ T₂ = ∀X. (T₁→T₂→X) → X;
```

correspond to pairs of $\mathtt{T_1}$ and $\mathtt{T_2}$. The constructor `pair` and the destructors `fst` and `snd` were defined as follows:

```
pair = λX. λY. λx:X. λy:Y.
         (   λR. λp:X→Y→R. p x y
          as Pair X Y);

fst = λX. λY. λp: Pair X Y.
        p X (λx:X. λy:Y. x);

snd = λX. λY. λp: Pair X Y.
        p Y (λx:X. λy:Y. y);
```

Of course, the same encoding can be used in $F_{<:}$, since $F_{<:}$ contains all the features of System F. What is interesting, though, is that this encoding also has some natural subtyping properties. In fact, the expected subtyping rule for pairs

$$\frac{\Gamma \vdash \mathtt{S}_1 <: \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{Pair}\ \mathtt{S}_1\ \mathtt{S}_2 <: \mathtt{Pair}\ \mathtt{T}_1\ \mathtt{T}_2}$$

follows directly from the encoding.

14.3.1.1 Exercise [Quick check]: Show this. □

14.3.2 Encoding Records

It is interesting to notice that records and record types—including subtyping—can actually be encoded in the pure calculus. The encoding presented here was discovered by Cardelli [Car92]. We begin by defining **flexible tuples** as follows:

14.3.2.1 Definition: For each $n \geq 0$ and types $\mathtt{T}_1$ through $\mathtt{T}_n$, let

$$\{\mathtt{T}_i\ ^{i \in 1..n}\} \stackrel{\text{def}}{=} \mathtt{Pair}\ \mathtt{T}_1\ (\mathtt{Pair}\ \mathtt{T}_2\ \ldots\ (\mathtt{Pair}\ \mathtt{T}_n\ \mathtt{Top})\ldots).$$

In particular, $\{\} = \mathtt{Top}$. Similarly, for terms $\mathtt{t}_1$ through $\mathtt{t}_n$, let

$$\{\mathtt{t}_i\ ^{i \in 1..n}\} \stackrel{\text{def}}{=} \mathtt{pair}\ \mathtt{t}_1\ (\mathtt{pair}\ \mathtt{t}_2\ \ldots\ (\mathtt{pair}\ \mathtt{t}_n\ \mathtt{top})\ldots),$$

where we elide the type arguments to `pair`, for the sake of brevity. (Recall that `top` is just some element of `Top`.) The projection `t.n` (again eliding type arguments) is:

$$\mathtt{fst}(\underbrace{\mathtt{snd}(\mathtt{snd}\ldots(\mathtt{snd}}_{n-1\ \text{times}}\mathtt{t})\ldots)$$ □

From this abbreviation, we immediately obtain the following rules for subtyping and typing.

$$\frac{\Gamma \vdash^{i \in 1..n} \mathtt{S}_i <: \mathtt{T}_i}{\Gamma \vdash \{\mathtt{S}_i\ ^{i \in 1..n+k}\} <: \{\mathtt{T}_i\ ^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash^{i \in 1..n} \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{t}_i\ ^{i \in 1..n}\} : \{\mathtt{T}_i\ ^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash \texttt{t} : \{\texttt{T}_i{}^{i\in 1..n}\}}{\Gamma \vdash \texttt{t.i} : \texttt{T}_i}$$

Now, let $\mathcal{L}$ be a countable set of labels, with a fixed total ordering given by the bijective function *label-with-index* $: \mathbb{N} \to \mathcal{L}$. We define records as follows:

14.3.2.2 Definition: Let L be a finite subset of $\mathcal{L}$ and let $\texttt{S}_l$ be a type for each $l \in L$. Let m be the maximal index of any element of L, and

$$\hat{\texttt{S}}_i = \begin{cases} \texttt{S}_l & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{Top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record type $\{l\texttt{:}\texttt{S}_l{}^{l\in L}\}$ is defined as the flexible tuple $\{\hat{\texttt{S}}_i^{\,i\in 1..m}\}$. Similarly, if $\texttt{t}_l$ is a term for each $l : L$, then

$$\hat{\texttt{t}}_i = \begin{cases} \texttt{t}_i & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record value $\{l\texttt{=}\texttt{t}_l{}^{l\in L}\}$ is $\{\hat{\texttt{t}}_i^{\,i\in 1..m}\}$. The projection $\texttt{t.l}_i$ is just the tuple projection $\texttt{t.i}$. $\square$

This encoding validates the expected rules for typing and subtyping:

$$\Gamma \vdash \{\texttt{l}_i\texttt{:}\texttt{T}_i{}^{i\in 1..n+k}\} <: \{\texttt{l}_i\texttt{:}\texttt{T}_i{}^{i\in 1..n}\} \qquad \text{(S-RCD-WIDTH)}$$

$$\frac{\Gamma \vdash \texttt{S}_i <: \texttt{T}_i}{\Gamma \vdash \{\texttt{l}_i\texttt{:}\texttt{S}_i{}^{i\in 1..n}\} <: \Gamma \vdash \{\texttt{l}_i\texttt{:}\texttt{T}_i{}^{i\in 1..n}\}} \qquad \text{(S-RCD-DEPTH)}$$

14.3.3 Church Encodings with Subtyping

As a last simple illustration of the expressiveness of $F_{<:}$, let's take a look at what happens when we add bounded quantification to the encoding of Church Numerals in System F that we saw in Section 12.4.3. The original polymorphic type of church numerals was:

```
CNat = ∀X. (X→X) → X → X;
```

The intuitive reading of this type was: "Tell me a result type `T` and give me a function on `T` and an element of `T`, and I'll give you back another element of `T` formed by iterating the function you gave me n times over the base value you gave."

We can generalize this by adding two bounded quantifiers and refining the types of the parameters `s` and `z`.

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

Intuitively, this type can be read as follows: "Give me a generic result type `T` and two subtypes `S` and `Z`. Then give me a function that maps from the whole set `T` into the subset S and an element of the special set Z, and I'll return you an element of T formed in the same way as before."

To see why this is an interesting generalization, consider this slightly different type:

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
```

Although SZero has almost the same form as SNat, it says something much stronger about the behavior of its elements, since it promises that its result will be an element of Z, not just of T. In fact, there is just one way that an element of Z could be returned—namely by yielding just z itself. In other words, the value

```
szero = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z) as SZero;
▸ szero : SZero
```

is the *only* inhabitant of the type SZero. On the other hand, the similar type

```
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

has more inhabitants; for example,

```
sone   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z) as SPos;
stwo   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z)) as SPos;
sthree = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z))) as SPos;
```

and so on.

Moreover, notice that SZero and SPos are both subtypes of SNat (Exercise: check this), so we also have zero : SNat, one : SNat, two : SNat, etc.

Finally, we can similarly refine the typings of operations defined on church numerals. For example, the type system is capable of detecting that the successor function always returns a positive number:

```
ssucc = λn:SNat.
          (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
            s (n [X] [S] [Z] s z))
          as SPos;
▸ ssucc : SNat → SPos
```

Similarly, by refining the types of its parameters, we can write the function plus in such a way that the typechecker gives it the refined type SPos→SZero→SPos.

```
spluspz = λn:SPos. λm:SZero.
            (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
              n [X] [S] [Z] s (m [X] [S] [Z] s z))
            as SPos;
▸ spluspz : SPos → SZero → SPos
```

14.3.3.1 Exercise: Write a similar version of plus that has type SPos→SPos→SPos. Write one that has type SNat→SNat→SNat. □

The previous example and exercise raise an interesting point: obviously, we don't want to have several different versions of `plus` lying around and have to decide which to apply based on the expected types of its arguments: we want to have a *single* version of `plus` whose type contains all these possibilities—something like

```
plus :   SZero→SZero→SZero
       ∧ SNat→SPos→SPos
       ∧ SPos→SNat→SPos
       ∧ SNat→SNat→SNat
```

The desire to support this kind of overloading has led to the study of systems with **intersection types**.

14.3.3.2 Exercise [Recommended]: Generalize the type `CBool` of Church Booleans from Section 12.4.3 in a similar way by defining a type `SBool` and two subtypes `STrue` and `SFalse`. Write a function `notft` with type `SFalse→STrue` and a similar one `nottf` with type `STrue→SFalse`. □

The examples that we have seen in this section are amusing to play with, but they might not convince you that $F_{<:}$ is a system of tremendous practical importance! We will come to some more interesting uses of bounded quantification in Chapter **??**, but these will require just a little more machinery, which we will develop in the intervening chapters.

14.4 Metatheory

We now consider the metatheory of both kernel and full systems with bounded quantification ($F^k_{<:}$ and $F^f_{<:}$). Much of the development is the same for both systems: we carry it out first for the simpler case of $F^k_{<:}$ and then consider $F^f_{<:}$.

14.4.1 Subject Reduction

The subject reduction property can actually be proved quite directly for both systems, with minimal technical preliminaries. This is good, since the soundness of the type system is a critical property, while other properties such as decidability may be less important in some contexts. We develop the proof in detail for $F^k_{<:}$. (The argument for $F^f_{<:}$ is similar.)

We begin with a couple of technical facts about the typing and subtyping relations. The proofs go by straightforward induction on derivations.

14.4.1.1 Lemma [Permutation]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{S} <: \mathtt{T}$. □

14.4.1.2 Lemma [Weakening]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin dom(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{X} \notin dom(\Gamma)$, then $\Gamma, \mathtt{X}{<:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
3. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{x} \notin dom(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$.
4. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{X} \notin dom(\Gamma)$, then $\Gamma, \mathtt{X}{<:}\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$. □

As usual, the proof of subject reduction relies on several lemmas relating substitution with the typing and subtyping relations.

14.4.1.3 Definition: We write $\{\mathtt{X} \mapsto \mathtt{S}\}\Gamma$ for the context obtained by substituting $\mathtt{S}$ for $\mathtt{X}$ in the right-hand sides of all of the bindings in Γ. □

14.4.1.4 Exercise [Quick check]: Show the following properties of subtyping and typing derivations:

1. if $\Gamma, \mathtt{X}{<:}\mathtt{U}, \Delta \vdash \mathtt{A} <: \mathtt{B}$ and $\Gamma \vdash \mathtt{C} <: \mathtt{U}$, then $\Gamma, \mathtt{X}{<:}\mathtt{C}, \Delta \vdash \mathtt{A} <: \mathtt{B}$;
2. if $\Gamma, \mathtt{X}{<:}\mathtt{U}, \Delta \vdash \mathtt{a} : \mathtt{A}$ and $\Gamma \vdash \mathtt{C} <: \mathtt{U}$, then $\Gamma, \mathtt{X}{<:}\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{A}$;

These properties are often called **narrowing** because they involve narrowing the type of the variable $\mathtt{X}$. They can be viewed as a form of strengthening. □

Next, we have the usual lemma relating substitution and the typing relation.

14.4.1.5 Lemma [Substitution preserves typing]: If $\Gamma, \mathtt{x}{:}\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{A}$ and $\Gamma \vdash \mathtt{c} : \mathtt{C}$, then $\Gamma, \Delta \vdash \{\mathtt{x} \mapsto \mathtt{c}\}\mathtt{a} : \mathtt{A}$. □

Proof: Straightforward induction on a derivation of $\Gamma, \mathtt{x}{:}\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{A}$, using the properties proved above. □

Since we may substitute types for type variables during reduction, we also need a lemma relating type substitution and typing, as we did in System F. Here, though, we must deal with one new twist: the proof of this lemma (specifically, the T-SUB case) depends on a new lemma relating substitution and subtyping:

14.4.1.6 Lemma [Type substitution preserves subtyping]: If $\Gamma, \mathtt{X}{<:}\mathtt{C}, \Delta \vdash \mathtt{D} <: \mathtt{A}$ and $\Gamma \vdash \mathtt{B} <: \mathtt{C}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{D} <: \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{A}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}{<:}\mathtt{C}, \Delta \vdash \mathtt{D} <: \mathtt{A}$. The only interesting cases are the last two:

Case S-TVAR*:* $\quad \mathtt{D} = \mathtt{Y} \qquad (\Gamma, \mathtt{X}{<:}\mathtt{C}, \Delta)(\mathtt{Y}) = \mathtt{Y}{<:}\mathtt{A}$

There are two subcases to consider. If $\mathtt{Y} \neq \mathtt{X}$, then the result follows immediately from S-TVAR. On the other hand, if $\mathtt{Y} = \mathtt{X}$, then we have $\mathtt{A} = \mathtt{C}$ and $\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{D} = \mathtt{C}$, and the result follows by S-REFL.

Case S-ALL: $\quad \mathtt{D} = \forall \mathtt{Z}<:\mathtt{U}.\mathtt{S} \quad \mathtt{A} = \forall \mathtt{Z}<:\mathtt{U}.\mathtt{T} \quad \Gamma, \mathtt{X}<:\mathtt{C}, \Delta, \mathtt{Z}<:\mathtt{U} \vdash \mathtt{S} <: \mathtt{T}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta, \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{U} \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{T}$. By S-ALL, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{U}.\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{S} <: \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{U}.\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{T}$, that is, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}(\forall \mathtt{Z}<:\mathtt{U}.\mathtt{S}) <: \{\mathtt{X} \mapsto \mathtt{B}\}(\forall \mathtt{Z}<:\mathtt{U}.\mathtt{T})$, as required. □

14.4.1.7 Lemma [Type substitution preserves typing]: If $\Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{A}$ and $\Gamma \vdash \mathtt{B} <: \mathtt{C}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{a} : \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{A}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{A}$. We give just the interesting cases.

Case T-TAPP: $\quad \mathtt{a} = \mathtt{t}\ \mathtt{S} \quad \mathtt{A} = \{\mathtt{Z} \mapsto \mathtt{S}\}\mathtt{T} \quad \Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash \mathtt{t} : \forall \mathtt{Z}<:\mathtt{U}.\mathtt{T}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{B}\}(\forall \mathtt{Z}<:\mathtt{U}.\mathtt{T})$, i.e, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{t} : \forall \mathtt{Z}<:\mathtt{U}.\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{T}$. By T-TAPP, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{t}\ (\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{S}) : \{\mathtt{Z} \mapsto \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{S}\}(\{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{T})$, i.e., $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}(\mathtt{t}\ \mathtt{S}) : \{\mathtt{X} \mapsto \mathtt{B}\}(\{\mathtt{Z} \mapsto \mathtt{S}\}\mathtt{T})$.

Case T-SUB: $\quad \Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash \mathtt{a} : \mathtt{D} \quad \Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash \mathtt{D} <: \mathtt{A}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{a} : \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{A}$. By the preservation of subtyping under substitution (14.4.1.6), $\Gamma, \{\mathtt{X} \mapsto \mathtt{B}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{D} <: \{\mathtt{X} \mapsto \mathtt{B}\}\mathtt{A}$, and the result follows by T-SUB. □

Next, we establish some simple structural facts about the subtype relation.

14.4.1.8 Lemma [Inversion of the subtyping relation, from right to left]:

1. If $\Gamma \vdash \mathtt{S} <: \mathtt{X}$, then $\mathtt{S}$ is a type variable.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}_1 \to \mathtt{T}_2$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S} = \mathtt{S}_1 \to \mathtt{S}_2$ with $\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1$ and $\Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2$.
3. If $\Gamma \vdash \mathtt{S} <: \forall \mathtt{X}<:\mathtt{U}.\mathtt{T}_2$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S} = \forall \mathtt{X}<:\mathtt{U}.\mathtt{S}_2$ with $\Gamma, \mathtt{X}<:\mathtt{U} \vdash \mathtt{S}_2 <: \mathtt{T}_2$. □

Proof: Part (1) follows by an easy induction on typing derivations. (The only interesting case is the rule S-TRANS, which proceeds by two uses of the induction hypothesis, first on the right premise and then on the left.) The arguments for the other parts are similar (part (1) is used in the transitivity cases). □

14.4.1.9 Exercise: Show the following "left to right inversion" properties:

1. If $\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}$, then either $\mathtt{T} = \mathtt{Top}$ or else $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$ with $\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1$ and $\Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2$.
2. If $\Gamma \vdash \forall \mathtt{X}<:\mathtt{U}.\mathtt{S}_2 <: \mathtt{T}$, then either $\mathtt{T} = \mathtt{Top}$ or else $\mathtt{T} = \forall \mathtt{X}<:\mathtt{U}.\mathtt{T}_2$ with $\Gamma, \mathtt{X}<:\mathtt{U} \vdash \mathtt{S}_2 <: \mathtt{T}_2$.
3. If $\Gamma \vdash \mathtt{X} <: \mathtt{T}$, then either $\mathtt{T} = \mathtt{Top}$ or $\mathtt{T} = \mathtt{X}$ or $\Gamma \vdash \Gamma(\mathtt{X}) <: \mathtt{T}$.

4. If $\Gamma \vdash \texttt{Top} <: \texttt{T}$, then $\texttt{T} = \texttt{Top}$. □

We use Lemma 14.4.1.8 for one straightforward structural property of the typing relation used in the critical cases of the subject reduction proof.

14.4.1.10 Lemma:

1. If $\Gamma \vdash \lambda\texttt{x:S}'\texttt{.u} : \texttt{S}{\to}\texttt{T}$, then there is some type $\texttt{U}$ such that $\Gamma, \texttt{x:S}' \vdash \texttt{u} : \texttt{U}$ with $\Gamma \vdash \texttt{S} <: \texttt{S}'$ and $\Gamma \vdash \texttt{U} <: \texttt{T}$.
2. If $\Gamma \vdash \lambda\texttt{X<:S.u} : \forall\texttt{X<:S}'\texttt{.T}$, then $\texttt{S} = \texttt{S}'$ and there is some type $\texttt{U}$ such that $\Gamma, \texttt{X<:S} \vdash \texttt{u} : \texttt{U}$ with $\Gamma \vdash \texttt{U} <: \texttt{T}$. □

Proof: Straightforward induction on typing derivations, using Lemma 14.4.1.8 for the induction case (rule T-SUB). □

With all these facts in-hand, the actual proof of subject reduction is straightforward.

14.4.1.11 Theorem [Subject reduction]: If $\Gamma \vdash \texttt{a} : \texttt{A}$ and $\Gamma \vdash \texttt{a} \heartsuit^{1} \texttt{a}'$, then $\Gamma \vdash \texttt{a}' : \texttt{A}$. □

Proof: By induction on a derivation of $\Gamma \vdash \texttt{a} : \texttt{A}$. All of the cases are straightforward, using the facts established in the above lemmas.

Case T-VAR: $\texttt{a} = \texttt{x} \quad \Gamma(\texttt{x}) = \texttt{A}$

This case cannot actually arise, since we assumed $\Gamma \vdash \texttt{a} \heartsuit^{1} \texttt{a}'$ and there are no reduction rules for reducing a variable to anything else.

Case T-ABS: $\texttt{a} = \lambda\texttt{x:S.t} \quad \Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T} \quad \texttt{A} = \texttt{S}{\to}\texttt{T}$

By the definition of the reduction relation, it must be the case that $\Gamma, \texttt{x:S} \vdash \texttt{t} \heartsuit^{1} \texttt{t}'$. By the induction hypothesis, $\Gamma, \texttt{x:S} \vdash \texttt{t}' : \texttt{T}$, from which the desired result, $\Gamma \vdash \texttt{a}' : \texttt{A}$, follows by T-ABS.

Case T-APP: $\texttt{a} = \texttt{t s} \quad \Gamma \vdash \texttt{t} : \texttt{S}{\to}\texttt{T} \quad \texttt{A} = \texttt{T} \quad \Gamma \vdash \texttt{s} : \texttt{S}$

By the definition of the reduction relation, there are three subcases to consider:

Subcase: $\Gamma \vdash \texttt{t} \heartsuit^{1} \texttt{t}' \quad \texttt{a} = \texttt{t}'\ \texttt{s}$

Then the result follows from the induction hypothesis and T-APP.

Subcase: $\Gamma \vdash \texttt{s} \heartsuit^{1} \texttt{s}' \quad \texttt{a} = \texttt{t}\ \texttt{s}'$

Similar.

Subcase: $\mathtt{t} = \lambda\mathtt{x{:}S'.u}$ $\quad \mathtt{a} = \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{u}$

By Lemma 14.4.1.10, $\Gamma, \mathtt{x{:}S'} \vdash \mathtt{u} : \mathtt{U}$ with $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{S'}$ and $\Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{T}$. By the preservation of typing under substitution (Lemma 14.4.1.5), $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{u} : \mathtt{U}$, and $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{u} : \mathtt{T}$ follows by T-SUB.

Case T-TABS*:* $\mathtt{a} = \lambda\mathtt{X{<:}U.t}$ $\quad \Gamma, \mathtt{X{<:}U} \vdash \mathtt{t} : \mathtt{T}$ $\quad \mathtt{A} = \forall\mathtt{X{<:}U.T}$

By the definition of the reduction relation, it must be the case that $\Gamma, \mathtt{X{<:}U} \vdash \mathtt{t} \heartsuit^{1} \mathtt{t'}$. The result then follows by the induction hypothesis and T-TABS.

Case T-TAPP*:* $\mathtt{a} = \mathtt{t\ S}$ $\quad \Gamma \vdash \mathtt{t} : \forall\mathtt{X{<:}W.T}$ $\quad \mathtt{A} = \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T}$ $\quad \Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{W}$

By the definition of the reduction relation, there are two subcases to consider:

Subcase: $\Gamma, \mathtt{X{<:}U} \vdash \mathtt{t} \heartsuit^{1} \mathtt{t'}$ $\quad \mathtt{a} = \mathtt{t'\ S}$

The result follows from the induction hypothesis and T-TAPP.

Subcase: $\mathtt{t} = \lambda\mathtt{X{<:}W.u}$ $\quad \mathtt{a} = \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{u}$

By Lemma 14.4.1.10, $\Gamma, \mathtt{X{<:}W} \vdash \mathtt{u} : \mathtt{U}$ with $\Gamma, \mathtt{X{<:}W} \vdash \mathtt{U} \mathrel{<:} \mathtt{T}$. By the preservation of typing under substitution (14.4.1.5), $\Gamma \vdash \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{u} : \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{U}$, and $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{u} : \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T}$ follows by Lemma 14.4.1.6 and T-SUB.

Case T-SUB*:* $\Gamma \vdash \mathtt{a} : \mathtt{D}$ $\quad \Gamma \vdash \mathtt{D} \mathrel{<:} \mathtt{A}$

By the induction hypothesis, $\Gamma \vdash \mathtt{a'} : \mathtt{D}$; the result follows by T-SUB. □

14.4.1.12 Exercise: Show how to extend the argument in this section to $F^{f}_{<:}$. □

14.4.2 Typing

Next we consider the problem of building a typechecking algorithm for a language with bounded quantifiers. The algorithm that we construct will be parametric in an algorithm for the subtype relation, which we consider in the following section.

Promotion

In the typechecking algorithm for $\lambda_{<:}$ in Section 10.3.2, the key idea was that we can calculate a *minimal* type for each term from the minimal types of its subterms. We will use the same basic idea to typecheck $F^{k}_{<:}$, but we need to consider one slight complication arising from the presence of type variables in the system.

Consider the term

```
f = λX<:Nat→Nat. λy:X. y 5;
▸ f : ∀X<:Nat→Nat. X → Nat
```

This term is clearly well typed, since the type of the variable `y` in the application `(y 5)` is `X`, can be promoted to `Nat`→`Nat` by T-SUB. But the *minimal* type of `y` is not an arrow type. In order to find the minimal type of the application, we need to find the minimal arrow type that `y` possesses—i.e., the minimal arrow type that is a supertype of `X`. It turns out that the correct way to find this type is to **promote** the minimal type of `y` until it is something other than a type variable.

Formally, write $\Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}$ to mean "`T` is the *least nonvariable supertype* of `S`," defined by repeated promotion of variables as follows:

$$\frac{\mathtt{S}\text{ is not a variable}}{\Gamma \vdash \mathtt{S} \Uparrow \mathtt{S}}$$

$$\frac{\Gamma(\mathtt{X}) = \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}}{\Gamma \vdash \mathtt{X} \Uparrow \mathtt{T}}$$

It is easy to check that these rules define a total function. Moreover, the result of promotion is always the least supertype that has some shape other than a variable.

14.4.2.1 Lemma: Suppose $\Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}$.

1. $\Gamma \vdash \mathtt{S} <: \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{U}$ and `U` is not a variable, then $\Gamma \vdash \mathtt{T} <: \mathtt{U}$. □

Proof: Part (1) is easy. Part (2) goes by straightforward induction on a derivation of $\Gamma \vdash \mathtt{S} <: \mathtt{U}$. □

Minimal Typing Algorithm

The algorithm for calculating minimal types is build along the same basic lines as the one for $\lambda_{<:}$, with one additional twist: the minimal type of a term may always be a type variable, which will need to be promoted to its smallest nonvariable supertype (it's smallest **concrete** supertype, we might say) in order to be used on the left of an application or type application.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash^{a} \mathtt{x} : \mathtt{T}} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash^{a} \mathtt{t}_1 : \mathtt{T}_2}{\Gamma \vdash^{a} \lambda \mathtt{x}{:}\mathtt{T}_1 . \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \vdash^{a} \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \vdash^{a} \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_1 \Uparrow \mathtt{U} \to \mathtt{V} \qquad \Gamma \vdash \mathtt{T}_2 <: \mathtt{U}}{\Gamma \vdash^{a} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{V}} \qquad \text{(TA-APP)}$$

$$\frac{\Gamma, \mathtt{X}{<:}\mathtt{U} \vdash^{a} \mathtt{t} : \mathtt{T}}{\Gamma \vdash^{a} \lambda \mathtt{X}{<:}\mathtt{U} . \mathtt{t} : \forall \mathtt{X}{<:}\mathtt{U} . \mathtt{T}} \qquad \text{(TA-TABS)}$$

$$\frac{\Gamma \vdash^{a} \mathtt{t_1} : \mathtt{T_1} \qquad \Gamma \vdash \mathtt{T_1} \Uparrow \forall\mathtt{X<:U.V} \qquad \Gamma \vdash \mathtt{T_2} <: \mathtt{U}}{\Gamma \vdash^{a} \mathtt{t_1}\ \mathtt{T_2} : \{\mathtt{X} \mapsto \mathtt{U}\}\mathtt{V}} \quad \text{(TA-TAPP)}$$

The proofs of soundness and completeness of this algorithm with respect to the original typing rules are fairly routine.

14.4.2.2 Theorem [Minimal typing]:

1. If $\Gamma \vdash^{a} \mathtt{a} : \mathtt{A}$, then $\Gamma \vdash \mathtt{a} : \mathtt{A}$.
2. If $\Gamma \vdash \mathtt{a} : \mathtt{A}$, then $\Gamma \vdash^{a} \mathtt{a} : \mathtt{M}$ where $\Gamma \vdash \mathtt{M} <: \mathtt{A}$. □

Proof: Part (1) proceeds by a straightforward induction on algorithmic derivations.

Part (2) is more interesting: it goes by induction on a derivation of $\Gamma \vdash \mathtt{a} : \mathtt{A}$. The most important cases are those for the rules T-APP and T-TAPP.

Case T-VAR: $\quad \mathtt{a} = \mathtt{x} \qquad \Gamma(\mathtt{x}) = \mathtt{T}$

By TA-VAR, $\Gamma \vdash^{a} \mathtt{x} : \mathtt{T}$. By S-REFL, $\Gamma \vdash \mathtt{T} <: \mathtt{T}$.

Case T-ABS: $\quad \mathtt{a} = \lambda\mathtt{x:S.t} \qquad \Gamma, \mathtt{x:S} \vdash \mathtt{t} : \mathtt{T}$
$\quad \mathtt{A} = \mathtt{S}\to\mathtt{T}$

By the induction hypothesis, $\Gamma, \mathtt{x:S} \vdash^{a} \mathtt{t} : \mathtt{N}$ for some $\mathtt{N}$ with $\Gamma, \mathtt{x:S} \vdash \mathtt{N} <: \mathtt{T}$, i.e., $\Gamma \vdash \mathtt{N} <: \mathtt{T}$. By TA-ABS, $\Gamma \vdash \mathtt{a} : \mathtt{S}\to\mathtt{N}$. Finally, by S-REFL and S-ARROW, we have $\Gamma \vdash \mathtt{S}\to\mathtt{N} <: \mathtt{S}\to\mathtt{T}$.

Case T-APP: $\quad \mathtt{a} = \mathtt{t}\ \mathtt{s} \qquad \Gamma \vdash \mathtt{t} : \mathtt{S}\to\mathtt{A} \qquad \Gamma \vdash \mathtt{s} : \mathtt{S}$

By the induction hypothesis, we have $\Gamma \vdash^{a} \mathtt{t} : \mathtt{M_t}$ and $\Gamma \vdash^{a} \mathtt{s} : \mathtt{M_s}$, with $\Gamma \vdash \mathtt{M_t} <: \mathtt{S}\to\mathtt{A}$ and $\Gamma \vdash \mathtt{M_s} <: \mathtt{S}$. Let $\mathtt{M}'$ be the least nonvariable supertype of $\mathtt{M_t}$—i.e., suppose $\Gamma \vdash \mathtt{M_t} \Uparrow \mathtt{M}'$. By the promotion lemma (14.4.2.1), $\Gamma \vdash \mathtt{M}' <: \mathtt{S}\to\mathtt{A}$. But we know that $\mathtt{M}'$ is not a variable, so the inversion lemma for the subtype relation (14.4.1.8) tells us that $\mathtt{M}' = \mathtt{M_1}\to\mathtt{M_2}$, with $\Gamma \vdash \mathtt{S} <: \mathtt{M_1}$ and $\Gamma \vdash \mathtt{M_2} <: \mathtt{A}$. By transitivity, $\Gamma \vdash \mathtt{M_s} <: \mathtt{S}$, and rule TA-APP gives us $\Gamma \vdash^{a} \mathtt{t}\ \mathtt{s} : \mathtt{M_2}$, as required.

Case T-TABS: $\quad \mathtt{a} = \lambda\mathtt{X<:U.t} \qquad \Gamma, \mathtt{X<:U} \vdash \mathtt{t} : \mathtt{T} \qquad \mathtt{A} = \forall\mathtt{X<:U.T}$

By the induction hypothesis, $\Gamma, \mathtt{X<:U} \vdash^{a} \mathtt{t} : \mathtt{N}$ for some $\mathtt{N}$ with $\Gamma, \mathtt{X<:U} \vdash \mathtt{N} <: \mathtt{T}$. By TA-TABS, $\Gamma \vdash^{a} \mathtt{a} : \forall\mathtt{X<:U.N}$. Finally, by S-REFL and S-ALL, we have $\Gamma \vdash \forall\mathtt{X<:U.N} <: \forall\mathtt{X<:U.T}$.

Case T-TAPP: $\quad \mathtt{a} = \mathtt{t}\ \mathtt{S} \qquad \Gamma \vdash \mathtt{t} : \forall\mathtt{X<:U.T} \qquad \mathtt{A} = \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{U}$

By the induction hypothesis, we have $\Gamma \vdash^{a} \mathtt{t} : \mathtt{M_t}$, with $\Gamma \vdash \mathtt{M_t} <: \forall\mathtt{X<:U.A}$. Let $\mathtt{M}'$ be the least nonvariable supertype of $\mathtt{M_t}$—i.e., suppose $\Gamma \vdash \mathtt{M_t} \Uparrow \mathtt{M}'$. By the promotion lemma (14.4.2.1), $\Gamma \vdash \mathtt{M}' <: \forall\mathtt{X<:U.A}$. But we know that $\mathtt{M}'$ is not a variable, so the inversion lemma for the subtype relation (14.4.1.8) tells us that $\mathtt{M}' = \forall\mathtt{X<:U.M_2}$, with $\Gamma, \mathtt{X<:U} \vdash \mathtt{M_2} <: \mathtt{T}$. Rule TA-TAPP gives us $\Gamma \vdash^{a} \mathtt{t}\ \mathtt{S} : \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{M_2}$, and the preservation of subtyping under substitution (14.4.1.6) yields $\Gamma \vdash \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{M_2} <: \{\mathtt{X} \mapsto \mathtt{S}\}\mathtt{T} = \mathtt{A}$.

Case T-SUB*:* $\Gamma \vdash$ `a : S` $\Gamma \vdash$ `S <: A`

By the induction hypothesis, $\Gamma \vdash^a$ `a : M` with $\Gamma \vdash$ `M <: S`. By S-TRANS, $\Gamma \vdash$ `M <: A`, from which T-SUB yields the desired result. □

14.4.2.3 Corollary [Decidability of typing]: The $F^k_{<:}$ typing relation is decidable, given a decision procedure for the subtyping relation. □

Proof: Given `t` and `T`, we can check whether there is some `T` such that $\Gamma \vdash$ `t : T` by using the algorithmic typing rules to check whether $\Gamma \vdash^a$ `t : T`. If so, then this `T` is also a type for `T` in the original typing relation (by part (1) of 14.4.2.2). If not, then part (2) of 14.4.2.2 implies that `t` has no type in the original typing relation.

Finally, note that the algorithmic typing rules constitute a terminating algorithm, since they are syntax-directed and always reduce the size of `t` when read from bottom to top. □

14.4.2.4 Exercise: Show how to add primitive booleans and conditionals as defined in Section **??** to the minimal typing algorithm for $F^k_{<:}$. (Solution on page **??**.) □

14.4.3 Subtyping in $F^k_{<:}$

As we saw in the simply typed lambda-calculus with subtyping, the subtyping rules in their present form do not constitute an algorithm for deciding the subtyping relation. We cannot use them "from bottom to top," for two reasons:

1. There are some overlaps between the conclusions of different rules (specifically, between S-REFL and nearly all the other rules). That is, looking at the form of a subtyping statement $\Gamma \vdash$ `S <: T`, we cannot decide which of the rules must have been used last in deriving it, if it is valid.

2. More seriously, one rule (S-TRANS) contains a metavariable in the premises that does not appear in the conclusion. To apply this rule from botom to top, we'd need to guess what type to replace this metavariable with.

The overlap between S-REFL and the other rules is easily dealt with, using exactly the same technique as we used in Chapter 10: we remove the full reflexivity rule and replace it by a restricted reflexivity rule

$$\Gamma \vdash \texttt{X <: X} \qquad \text{(S-REFL-TVAR)}$$

that applies only to type variables.

Next we must deal with S-TRANS. Unfortunately, unlike the simple subtyping relation studied in Chapter 10, the transitivity rule here interacts in an important way with another rule—S-TVAR, which allows assumptions about type variables to be used in deriving subtyping statements. For example, if

$$\Gamma \quad = \quad \texttt{W<:Top, X<:W, Y<:X, Z<:Y}$$

then the statement

$\Gamma \vdash \mathtt{Z} <: \mathtt{W}$

is provable using all the subtyping rules, but cannot be proved if S-TRANS is removed. That is, an instance of S-TRANS whose left-hand subderivation is an instance of the axiom S-TVAR, as in

$$\dfrac{\dfrac{}{\Gamma \vdash \mathtt{Z} <: \mathtt{Y}}\ (\text{S-TVAR}) \qquad \dfrac{\vdots}{\Gamma \vdash \mathtt{Y} <: \mathtt{W}}}{\Gamma \vdash \mathtt{Z} <: \mathtt{W}}\ (\text{S-TRANS})$$

cannot, in general, be eliminated.

Fortunately, it turns out that derivations of this form are the *only* essential uses of transitivity in subtyping. This informal observation can be made precise by introducing a new subtyping rule

$$\frac{\Gamma \vdash \Gamma(\mathtt{X}) <: \mathtt{T}}{\Gamma \vdash \mathtt{X} <: \mathtt{T}} \qquad \text{(SA-TRANS-TVAR)}$$

that captures precisely this pattern of variable lookup followed by transitivity.

Subtyping Algorithm

The intuitions in the previous section are summarized in the following definition. The algorithmic subtype relation of $F^k_{<:}$ is the least relation closed under the following rules:

$$\Gamma \vdash^a \mathtt{X} <: \mathtt{X} \qquad \text{(SA-REFL-TVAR)}$$

$$\frac{\Gamma(\mathtt{X}) = \mathtt{U} \qquad \Gamma \vdash^a \mathtt{U} <: \mathtt{T}}{\Gamma \vdash^a \mathtt{X} <: \mathtt{T}} \qquad \text{(SA-TRANS-TVAR)}$$

$$\Gamma \vdash^a \mathtt{S} <: \mathtt{Top} \qquad \text{(SA-TOP)}$$

$$\frac{\Gamma \vdash^a \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash^a \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash^a \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(SA-ARROW)}$$

$$\frac{\Gamma, \mathtt{X}<:\mathtt{U} \vdash^a \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash^a \forall \mathtt{X}<:\mathtt{U}.\mathtt{S}_2 <: \forall \mathtt{X}<:\mathtt{U}.\mathtt{T}_2} \qquad \text{(SA-ALL)}$$

14.4.3.1 Lemma [Reflexibity of the algorithmic subtype relation]: The algorithmic statement $\Gamma \vdash^a \mathtt{A} <: \mathtt{A}$ is provable for all Γ and $\mathtt{A}$. □

Proof: Easy induction on $\mathtt{A}$. □

14.4.3.2 Lemma [Transitivity of the algorithmic subtype relation]: If $\Gamma \vdash^a \mathtt{A} <: \mathtt{B}$ and $\Gamma \vdash^a \mathtt{B} <: \mathtt{C}$, then $\Gamma \vdash^a \mathtt{A} <: \mathtt{C}$. □

Proof: By induction on the sum of the sizes of the two derivations. Given two derivations of some total size, we proceed by considering the final rules in each.

First, if the right-hand derivation is an instance of SA-TOP, then we are done, since $\Gamma \vdash \mathtt{A} <: \mathtt{Top}$ by SA-TOP. Moreover, if the left-hand derivation is an instance of SA-TOP, then $\mathtt{B} = \mathtt{Top}$ and by looking at the algorithmic rules we see that the right-hand derivation must also be an instance of SA-TOP.

If either derivation is an instance of SA-REFL-TVAR, then we are again done since the other derivation is the desired result.

Next, if the left-hand derivation ends with an instance of SA-TRANS-TVAR, then $\mathtt{A} = \mathtt{X}$ with $\Gamma(\mathtt{X}) = \mathtt{S}$ and we have a subderivation with conclusion $\Gamma \vdash \mathtt{S} <: \mathtt{B}$. By the induction hypothesis, $\Gamma \vdash^{a} \mathtt{S} <: \mathtt{C}$, and, by SA-TRANS-TVAR again, $\Gamma \vdash^{a} \mathtt{X} <: \mathtt{C}$, as required.

If the left-hand derivation ends with an instance of SA-ARROW, then we have $\mathtt{A} = \mathtt{A}_1 \to \mathtt{A}_2$ and $\mathtt{B} = \mathtt{B}_1 \to \mathtt{B}_2$, with subderivations $\Gamma \vdash^{a} \mathtt{B}_1 <: \mathtt{A}_1$ and $\Gamma \vdash^{a} \mathtt{A}_2 <: \mathtt{B}_2$. But, since we have already considered the case where the right-hand derivation is SA-TOP, the only remaining possibility is that the right-hand derivation also ends with SA-ARROW; we therefore have $\mathtt{C} = \mathtt{C}_1 \to \mathtt{C}_2$, and two more subderivations $\Gamma \vdash^{a} \mathtt{C}_1 <: \mathtt{B}_1$ and $\Gamma \vdash^{a} \mathtt{B}_2 <: \mathtt{C}_2$. We now apply the induction hypothesis twice, obtaining $\Gamma \vdash^{a} \mathtt{C}_1 <: \mathtt{A}_1$ and $\Gamma \vdash^{a} \mathtt{A}_2 <: \mathtt{C}_2$. Finally, SA-ARROW yields $\Gamma \vdash^{a} \mathtt{A}_1 \to \mathtt{A}_2 <: \mathtt{C}_1 \to \mathtt{C}_2$, as required.

The case where the left-hand derivation ends with an instance of SA-ALL is similar. □

14.4.3.3 Theorem [Soundness and completeness of algorithmic subtyping]:

1. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ then $\Gamma \vdash^{a} \mathtt{S} <: \mathtt{T}$.

2. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ then $\Gamma \vdash \mathtt{S} <: \mathtt{T}$. □

Proof: Both directions proceed by induction on derivations. Soundness is straightforward. Completeness is also straightforward, since we have already done the hard work (for the reflexivity and transitivity rules of the original subtype relation) in Lemmas 14.4.3.1 and 14.4.3.2. □

Finally, we should check that the subtyping rules define an algorithm that is *total*. We do this by assigning a weight to each subtyping statement, and checking that the algorithmic rules all have conclusions with greater weight than their premises.

14.4.3.4 Definition: The **weight** of a type $\mathtt{T}$ in a context Γ, written $weight_{\Gamma}(\mathtt{T})$, is defined as follows:

$$
\begin{array}{lcll}
weight_{\Gamma}(\mathtt{X}) & = & weight_{\Gamma_1}(\mathtt{U}) + 1 & \text{if } \Gamma = \Gamma_1, \mathtt{X}{<:}\mathtt{U}, \Gamma_2 \\
weight_{\Gamma}(\mathtt{Top}) & = & 1 & \\
weight_{\Gamma}(\mathtt{S}\to\mathtt{T}) & = & weight_{\Gamma}(\mathtt{S}) + weight_{\Gamma}(\mathtt{T}) + 1 & \\
weight_{\Gamma}(\forall\mathtt{X}{<:}\mathtt{U}.\mathtt{T}) & = & weight_{\Gamma,\,\mathtt{X}<:\mathtt{U}}(\mathtt{T}) + 1 &
\end{array}
$$

The **weight** of a subtyping statement "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$" is the maximum weight of $\mathtt{S}$ and $\mathtt{T}$ in Γ. □

14.4.3.5 Theorem: The weight of the conclusion in an instance of any of the algorithmic subtyping rules is strictly greater than the weight of any of the premises. □

Proof: Straightforward inspection of the rules. □

14.4.4 Subtyping in $F^f_{<:}$

The only difference in the full system $F^f_{<:}$ is that the quantifier subtyping rule S-ALL is replaced by the more expressive variant:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \\ \Gamma, \mathtt{X}<:\mathtt{T}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2\end{array}}{\Gamma \vdash \forall \mathtt{X}<:\mathtt{S}_1.\mathtt{S}_2 <: \forall \mathtt{X}<:\mathtt{T}_1.\mathtt{T}_2} \qquad \text{(S-ALL)}$$

Subtyping Algorithm

The algorithmic subtype relation of $F^f_{<:}$ consists of exactly the same set of rules as the algorithm for $F^k_{<:}$, except that SA-ALL is refined to reflect the new version of S-ALL:

$$\Gamma \vdash^a \mathtt{X} <: \mathtt{X} \qquad \text{(SA-REFL-TVAR)}$$

$$\frac{\Gamma(\mathtt{X}) = \mathtt{U} \qquad \Gamma \vdash^a \mathtt{U} <: \mathtt{T}}{\Gamma \vdash^a \mathtt{X} <: \mathtt{T}} \qquad \text{(SA-TRANS-TVAR)}$$

$$\Gamma \vdash^a \mathtt{S} <: \mathtt{Top} \qquad \text{(SA-TOP)}$$

$$\frac{\Gamma \vdash^a \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash^a \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash^a \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(SA-ARROW)}$$

$$\frac{\Gamma \vdash^a \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{X}<:\mathtt{T}_1 \vdash^a \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash^a \forall \mathtt{X}<:\mathtt{S}_1.\mathtt{S}_2 <: \forall \mathtt{X}<:\mathtt{T}_1.\mathtt{T}_2} \qquad \text{(SA-ALL)}$$

As with $F^k_{<:}$, the soundness and completeness of this algorithmic relation with respect to the original subtype relation can be shown easily, once we have established that the algorithmic relation is reflexive and transitive. For reflexivity, the argument is exactly the same as before. For transitivity, on the other hand, the issues are more subtle.

14.4.4.1 Lemma [Transitivity and narrowing]:

1. If $\Gamma \vdash^a \mathtt{A} <: \mathtt{B}$ and $\Gamma \vdash^a \mathtt{B} <: \mathtt{C}$, then $\Gamma \vdash^a \mathtt{A} <: \mathtt{C}$.
2. If $\Gamma, \mathtt{X}<:\mathtt{B}, \Delta \vdash^a \mathtt{D} <: \mathtt{E}$ and $\Gamma \vdash^a \mathtt{C} <: \mathtt{B}$ then $\Gamma, \mathtt{X}<:\mathtt{C}, \Delta \vdash^a \mathtt{D} <: \mathtt{E}$. □

Proof: The two parts are proved simultaneously, by induction on the size of B. At each stage of the induction, the argument for part (2) assumes that part (1) has been established already for the B in question; part (1), on the other hand, uses part (2) only for strictly smaller Bs.

1. Most of the argument is identical to the proof of 14.4.3.2. The only difference lies in the case where both of the given derivations end in instances of SA-ALL. In this case, we have

   $$\begin{array}{l} \mathtt{A} = \forall \mathtt{X}\mathtt{<:}\mathtt{A}_1.\mathtt{A}_2 \\ \mathtt{B} = \forall \mathtt{X}\mathtt{<:}\mathtt{B}_1.\mathtt{B}_2 \\ \mathtt{C} = \forall \mathtt{X}\mathtt{<:}\mathtt{C}_1.\mathtt{C}_2 \\ \Gamma \vdash \mathtt{B}_1 \mathrel{<:} \mathtt{A}_1 \\ \Gamma, \mathtt{X}\mathtt{<:}\mathtt{B}_1 \vdash^a \mathtt{A}_2 \mathrel{<:} \mathtt{B}_2 \Gamma \vdash \mathtt{C}_1 \mathrel{<:} \mathtt{B}_1 \\ \Gamma, \mathtt{X}\mathtt{<:}\mathtt{C}_1 \vdash^a \mathtt{B}_2 \mathrel{<:} \mathtt{A}_2 \end{array}$$

   as subderivations. By part (1) of the induction hypothesis, we can immediately combine the three subderivations for the bounds to obtain $\Gamma \vdash^a \mathtt{C}_1 \mathrel{<:} \mathtt{A}_1$. For the bodies, we need to work a little harder, since the two contexts do not quite agree. We first use part (2) of the induction hypothesis to "narrow" the bound of X in the derivation of $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{B}_1 \vdash^a \mathtt{A}_2 \mathrel{<:} \mathtt{B}_2$, obtaining $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{C}_1 \vdash^a \mathtt{A}_2 \mathrel{<:} \mathtt{B}_2$. Now part (1) of the induction hypothesis applies, yielding $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{C}_1 \vdash^a \mathtt{A}_2 \mathrel{<:} \mathtt{C}_2$. Finally, by SA-ALL, $\Gamma \vdash^a \forall \mathtt{X}\mathtt{<:}\mathtt{A}_1.\mathtt{A}_2 \mathrel{<:} \forall \mathtt{X}\mathtt{<:}\mathtt{C}_1.\mathtt{C}_2$, as required.

2. Given B of a certain size, we use an inner induction on the size of a derivation of $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{B}, \Delta \vdash^a \mathtt{D} \mathrel{<:} \mathtt{E}$, with a case analysis on the last rule used in this derivation.

   The only interesting case is SA-TRANS-TVAR, where $\mathtt{D} = \mathtt{X}$ and $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{B}, \Delta \vdash^a \mathtt{B} \mathrel{<:} \mathtt{E}$ as a subderivation. By the inner induction hypothesis, $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{C}, \Delta \vdash^a \mathtt{B} \mathrel{<:} \mathtt{E}$. Rule SA-TRANS-TVAR now yields $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{C}, \Delta \vdash^a \mathtt{X} \mathrel{<:} \mathtt{E}$, as required. □

14.4.4.2 Exercise: Show that $F^k_{<:}$ has joins and bounded meets. (Solution on page **??**.) □

14.4.4.3 Exercise: Consider the types (due to Ghelli [Ghe90, p. 92])

$$\mathtt{S} = \forall \mathtt{X}\mathtt{<:}\mathtt{A}{\to}\mathtt{B}.\mathtt{A}{\to}\mathtt{B}$$

and

$$\mathtt{T} = \forall \mathtt{X}\mathtt{<:}\mathtt{A}'{\to}\mathtt{B}'.\mathtt{A}'{\to}\mathtt{B}'$$

and the context $\Gamma = \mathtt{A}\mathtt{<:}\mathtt{Top}, \mathtt{B}\mathtt{<:}\mathtt{Top}, \mathtt{A}'\mathtt{<:}\mathtt{A}, \mathtt{B}'\mathtt{<:}\mathtt{B}$.

1. In $F^f_{<:}$, how many types are subtypes of both S and T under Γ.

2. Show that, in $F^f_{<:}$, the types S and T have no meet under Γ.

3. Exhibit a pair of types that has no join (under Γ) in $F^f_{<:}$.

(Solution on page **??**.) □

14.5 Undecidability of Subtyping in Full $F_{<:}$

Unfortunately, the proof of termination of the $F^k_{<:}$ subtyping algorithm in Section 14.4.3 does not work for $F^f_{<:}$.

14.5.1 Exercise [Quick check]: Why? □

In fact, not only does this particular proof technique not work—the subtyping algorithm actually does *not* terminate on some inputs. For example, here is an input, due to Ghelli [Ghe95], that makes the algorithm diverge. First define the following abbreviation:

$$\neg\mathtt{S} \quad \stackrel{\text{def}}{=} \quad \forall\mathtt{X}\texttt{<:}\mathtt{S}.\mathtt{X}.$$

14.5.2 Fact: $\Gamma \vdash \neg\mathtt{S} <: \neg\mathtt{T}$ iff $\Gamma \vdash \mathtt{T} <: \mathtt{S}$. □

Proof: Exercise. □

Now, define a type T as follows:

$$\mathtt{T} \quad = \quad \forall\mathtt{X}\texttt{<:}\mathtt{Top}.\ \neg(\forall\mathtt{Y}\texttt{<:}\mathtt{X}.\neg\mathtt{Y}).$$

If we use the algorithmic subtyping rules bottom-to-top to attempt to construct a subtyping derivation for the statement

$$\mathtt{X}_0\texttt{<:}\mathtt{T} \vdash \mathtt{X}_0 <: (\forall\mathtt{X}_1\texttt{<:}\mathtt{X}_0.\neg\mathtt{X}_1)$$

we end up in an infinite regress of larger and larger "subgoals":

$$\begin{array}{lcllcl}
\mathtt{X}_0\texttt{<:}\mathtt{T} & \vdash & \mathtt{X}_0 & <: & (\forall\mathtt{X}_1\texttt{<:}\mathtt{X}_0.\neg\mathtt{X}_1) \\
\mathtt{X}_0\texttt{<:}\mathtt{T} & \vdash & \forall\mathtt{X}_1\texttt{<:}\mathtt{Top}.\ \neg(\forall\mathtt{X}_2\texttt{<:}\mathtt{X}_1.\neg\mathtt{X}_2) & <: & (\forall\mathtt{X}_1\texttt{<:}\mathtt{X}_0.\neg\mathtt{X}_1) \\
\mathtt{X}_0\texttt{<:}\mathtt{T}, \mathtt{X}_1\texttt{<:}\mathtt{X}_0 & \vdash & \neg(\forall\mathtt{X}_2\texttt{<:}\mathtt{X}_1.\neg\mathtt{X}_2) & <: & \neg\mathtt{X}_1 \\
\mathtt{X}_0\texttt{<:}\mathtt{T}, \mathtt{X}_1\texttt{<:}\mathtt{X}_0 & \vdash & \mathtt{X}_1 & <: & (\forall\mathtt{X}_2\texttt{<:}\mathtt{X}_1.\neg\mathtt{X}_2) \\
\mathtt{X}_0\texttt{<:}\mathtt{T}, \mathtt{X}_1\texttt{<:}\mathtt{X}_0 & \vdash & \mathtt{X}_0 & <: & (\forall\mathtt{X}_2\texttt{<:}\mathtt{X}_1.\neg\mathtt{X}_2) \\
\text{etc.}
\end{array}$$

The α-conversion steps necessary to maintain the well-formedness of the context when new variables are added are performed tacitly here, choosing new names so as to clarify the pattern of regress. The crucial trick is the "re-bounding" that occurs, for instance, between the second and third lines, where the bound of $\mathtt{X}_1$

on the left-hand side is changed from Top in line 2 to X_0 in line 3. Since the whole left-hand side in line 2 is itself the upper-bound of X_0, the re-bounding creates a cyclic pattern where longer and longer chains of variables in the context must be traversed on each loop. (The reader is cautioned not to look for *semantic* intuitions behind this example; in particular, $\neg T$ is a negation only in the sense that it allows the left- and right-hand sides of subtyping judgements to be swapped.)

Worse yet, not only does this particular algorithm fail to terminate on some inputs, it can be shown [Pie94] that there is *no* algorithm that is sound and complete for the original $F^f_{<:}$ subtyping relation and that terminates on all inputs.

The full proof of this fact is beyond the scope of this book. (The argument requires no particularly deep mathematics, but takes several pages to develop.) However, to give a little of its flavor, let's look at one more example.

14.5.3 Definition: The **positive** and **negative** occurences in a type T are defined as follows:

- T itself is a positive occurence in T.
- If $T_1 \to T_2$ is a positive (respectively, negative) occurrence, then T_1 is a negative (resp. positive) occurrence and T_2 is a positive (negative) occurrence.
- If $\forall X<:T_1.T_2$ is a positive (respectively, negative) occurrence, then T_1 is a negative (resp. positive) occurrence and T_2 is a positive (negative) occurrence.

The positive and negative occurences in a subtyping statement $\Gamma \vdash S <: T$ are defined as follows: the type S and the bounds of type variables in Γ are negative occurrences. The type T is a positive occurrence. □

(The words "positive" and "negative" come from logic. According to the well-known "Curry-Howard isomorphism" [How80, CF58]between propositions and types, the type $S \to T$ corresponds to the logical proposition $S \Rightarrow T$, which, by the definition of logical implication, is equivalent to $\neg S \vee T$. The subproposition S here is obviously in a "negative" position—that is, inside of an odd number of negations—if and only if the whole implication appears inside an even number of negations.)

14.5.4 Fact: If X occurs only positively in S and negatively in T, then $X<:U \vdash S <: T$ iff $\vdash \{X \mapsto U\}S <: \{X \mapsto U\}T$. □

Proof: Exercise. □

Now, let T be the following (rather horrible) type

$$
\begin{aligned}
T = {} & \forall X_0<:Top.\forall X_1<:Top.\forall X_2<:Top. \\
& \quad \neg(\forall Y_0<:X_0.\forall Y_1<:X_1.\forall Y_2<:X_2.\neg X_0)
\end{aligned}
$$

and consider the subtyping statement

$$\begin{array}{ll} \vdash & \mathtt{T} \\ <: & \forall \mathtt{X}_0 \mathtt{<:T}.\forall \mathtt{X}_1\mathtt{<:P}.\forall \mathtt{X}_2\mathtt{<:Q}. \\ & \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. \\ & \qquad \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_1.\ \mathtt{U})). \end{array}$$

We can think of this statement as a description of the state of a simple computer:

- The variables $\mathtt{X}_1$ and $\mathtt{X}_2$ are the "registers" of this machine. Their current contents are the types P and Q.
- The "instruction stream" of the machine is the last line of the statement: the first instruction is encoded in the bounds ($\mathtt{Y}_2$ and $\mathtt{Y}_1$—note their order!) of the variables $\mathtt{Z}_1$ and $\mathtt{Z}_2$, and the type U is the remaining instructions in the program.
- The type T, the nested negations, and the bound variables $\mathtt{X}_0$ and $\mathtt{Y}_0$ here play much the same role as their counterparts in the simpler example above: they allow us to "turn the crank" and get back to a subgoal of the same shape as the original goal. One turn of the crank will correspond to one cycle of our machine.

In this example, the instruction at the front of the instruction stream encodes the command "switch the contents of registers 1 and 2." To see this, we use the two facts stated above to calculate as follows. (The values P and Q in the two registers are highlighted, to make them easier to track through the derivation.)

$$\begin{array}{llll}
 & \vdash & \mathtt{T} & \\
 & <: & \forall \mathtt{X}_0\mathtt{<:T}.\forall \mathtt{X}_1\mathtt{<:P}.\forall \mathtt{X}_2\mathtt{<:Q}. & \\
 & & \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \\
 & & \qquad \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_1.\ \mathtt{U})) & \\
\text{iff} & \vdash & \neg(\forall \mathtt{Y}_0\mathtt{<:T}.\forall \mathtt{Y}_1\mathtt{<:P}.\forall \mathtt{Y}_2\mathtt{<:Q}.\neg \mathtt{T}) & \\
 & <: & \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \text{by Fact 14.5.4} \\
 & & \quad \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_1.\ \mathtt{U})) & \\
\text{iff} & \vdash & (\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \\
 & & \quad \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_1.\ \mathtt{U})) & \\
 & <: & (\forall \mathtt{Y}_0\mathtt{<:T}.\forall \mathtt{Y}_1\mathtt{<:P}.\forall \mathtt{Y}_2\mathtt{<:Q}.\neg \mathtt{T}) & \text{by Fact 14.5.2} \\
\text{iff} & \vdash & \neg(\forall \mathtt{Z}_0\mathtt{<:T}.\forall \mathtt{Z}_1\mathtt{<:Q}.\forall \mathtt{Z}_2\mathtt{<:P}.\ \mathtt{U})) & \\
 & <: & \neg \mathtt{T} & \text{by Fact 14.5.4} \\
\text{iff} & \vdash & \mathtt{T} & \\
 & <: & (\forall \mathtt{Z}_0\mathtt{<:T}.\forall \mathtt{Z}_1\mathtt{<:Q}.\forall \mathtt{Z}_2\mathtt{<:P}.\ \mathtt{U})) & \text{by Fact 14.5.2}
\end{array}$$

Note that, at the end of the derivation, not only have the values P and Q switched places, but the instruction that caused this to happen has been used up in the process, leaving U at the front of the instruction stream to be "executed" next. By choosing a value of U that begins in the same way as the instruction we just executed

$$\mathtt{U} \;=\; \neg(\forall \mathtt{Y}_0\texttt{<:Top}.\forall \mathtt{Y}_1\texttt{<:Top}.\forall \mathtt{Y}_2\texttt{<:Top}.\ \neg(\forall \mathtt{Z}_0\texttt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\texttt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\texttt{<:}\mathtt{Y}_1.\ \mathtt{U}'))$$

we can perform another swap and return the registers to their original state before continuing with U′. More interestingly, we can choose other values for U that cause different sorts of behavior. For example, if

$$\mathtt{U} \;=\; \neg(\forall \mathtt{Y}_0\texttt{<:Top}.\forall \mathtt{Y}_1\texttt{<:Top}.\forall \mathtt{Y}_2\texttt{<:Top}.\ \neg(\forall \mathtt{Z}_0\texttt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\texttt{<:}\mathtt{Y}_1.\forall \mathtt{Z}_2\texttt{<:}\mathtt{Y}_2.\ \mathtt{Y}_1))$$

then, on the next cycle of the machine, the current value of register 1 (Q) will appear in the position of U—in effect, performing an "indirect branch" through register 1 to the stream of instructions Q. Branching constructs and arithmetic (successor, predecessor, and zero-test) can be encoded using a generalization of this trick.

Putting all of this together, we arrive at a proof of undecidability, via a reduction from two-counter machines (a simple variant on ordinary turing machines, consisting of a finite control and two tapes, each holding a natural number—cf. [HU79], for example) to subtyping statements.

14.5.5 Theorem [Undecidability]: For each two-counter machine M, there exists a subtyping statement $S(M)$ such that $S(M)$ is derivable in $F^f_{<:}$ iff the execution of M halts. □

Thus, if we could decide whether any subtype statement is provable, then we could also decide whether any given two-counter machine will eventually halt.

14.5.6 Exercise [Challenging]:

1. Define a variant of $F^f_{<:}$ with no Top type but with both X<:T and X bindings.
2. Show that the subtype relation for this system is decidable.
3. Does this restriction offer a satisfactory solution to the basic problem? In particular, does it work for languages with additional features such as numbers, records, variants, etc.?

(Solution on page **??**.) □

14.6 Bounded Existential Types

□ : **Extends**

New syntactic forms:

$$
\begin{array}{lll}
\texttt{t} ::= & \dots & \\
 & \texttt{\{}\exists\texttt{X=T,t\} as T} & \textit{packing} \\
 & \texttt{let \{X,x\}=t in t} & \textit{unpacking} \\
\texttt{v} ::= & \dots & \\
 & \texttt{\{}\exists\texttt{X=T,v\} as T} & \textit{package value} \\
\texttt{T} ::= & \dots & \\
 & \texttt{\{}\exists\texttt{X<:T,T\}} & \textit{existential type}
\end{array}
$$

New evaluation rules:

$$\begin{array}{c}\texttt{let \{X,x\}=\{}\exists\texttt{X=T}_{11}\texttt{,v}_{12}\texttt{\} in t}_2 \\ \longrightarrow \{\texttt{X} \mapsto \texttt{T}_{11}\}\{\texttt{x} \mapsto \texttt{v}_{12}\}\texttt{t}_{12}\end{array} \quad \text{(EV-PACKBETA)}$$

$$\frac{\texttt{t}_{12} \longrightarrow \texttt{t}'_{12}}{\begin{array}{c}\texttt{\{}\exists\texttt{X=T}_{11}\texttt{,t}_{12}\texttt{\} in t}_2 \\ \longrightarrow \texttt{\{}\exists\texttt{X=T}_{11}\texttt{,t}'_{12}\texttt{\} in t}_2\end{array}} \quad \text{(EV-PACK)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}'_1}{\begin{array}{c}\texttt{let \{X,x\}=t}_1\texttt{ in t}_2 \\ \longrightarrow \texttt{let \{X,x\}=t}'_1\texttt{ in t}_2\end{array}} \quad \text{(EV-UNPACK)}$$

New subtyping rules:

$$\frac{\Gamma, \texttt{X<:U} \vdash \texttt{S}_2 <: \texttt{T}_2}{\Gamma \vdash \texttt{\{}\exists\texttt{X<:U,S}_2\texttt{\}} <: \texttt{\{}\exists\texttt{X<:U,T}_2\texttt{\}}} \quad \text{(S-SOME)}$$

New typing rules:

$$\frac{\Gamma \vdash \texttt{t}_2 : \{\texttt{X} \mapsto \texttt{T}_1\}\texttt{T}_2 \qquad \Gamma \vdash \texttt{T} <: \texttt{T}_1}{\Gamma \vdash \texttt{\{}\exists\texttt{X=T}_1\texttt{,t}_2\texttt{\} as \{}\exists\texttt{X<:T,T}_2\texttt{\} : \{}\exists\texttt{X<:T,T}_2\texttt{\}}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \exists\texttt{X<:T.T}_{11} \qquad \Gamma, \texttt{X<:T}, \texttt{x:T}_{11} \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let \{X,x\}=t}_1\texttt{ in t}_2 : \texttt{T}_2} \quad \text{(T-UNPACK)}$$

14.6.1 Exercise: Show how the subtyping rule S-SOME can be obtained from the subtyping rules for universals by extending the encoding of existential types in terms of universal types described in Exercise **??**. □

14.7 Historical Notes and Further Reading

The idea of bounded quantification was introduced by Cardelli and Wegner [CW85] in the language Fun. (Their "Kernel Fun" calculus corresponds to our $F^k_{<:}$.) Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [Mit84b], Fun integrated Girard-Reynolds polymorphism [Gir72, Rey74] with Cardelli's first-order calculus of subtyping [**?**, Car84]. The original Fun was simplified and slightly generalized by Bruce and Longo [BL90], and again by Curien and Ghelli [CG92], yielding the calculus we call $F^f_{<:}$.

The most comprehensive single paper on bounded quantification is the survey by Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

Fun and its relatives have been studied extensively by programming language theorists and designers. Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [Car88]. Curien and Ghelli [CG92, Ghe90] address a number of syntactic properties of $F_{<:}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [BL90], Martini [Mar88], Breazu-Tannen, Coquand, Gunter, and Scedrov [BCGS91], Cardone [Car89], Cardelli and Longo [CL91], Cardelli, Martini, Mitchell, and Scedrov [**?**], Curien and Ghelli [CG92, CG91], and Bruce and Mitchell [BM92]. $F_{<:}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [CM91], Bruce [Bru91], Cardelli [Car92], and Canning, Cook, Hill, Olthoff, and Mitchell [CCH+89]. Bounded quantification also plays a key role in Cardelli's programming language Quest [Car91, CL91] and in the Abel language developed at HP Labs [CCHO89, CCH+89, CHO88, CHC90].

The undecidability of full $F_{<:}$ was shown by Pierce [Pie94] and further analyzed by Ghelli [Ghe95].

The effect of bounded quantification on Church encodings of algebraic datatypes (cf. Section 14.3.3) was considered by Ghelli thesis [Ghe90] and Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

An extension of $F_{<:}$ with intersection types was studied by Pierce [Pie91, Pie97] and applied to the modeling of object-oriented languages with multiple inheritance by Compagnoni and Pierce [CP96].

Appendices

Appendix A

Solutions to Selected Exercises

Solution to 4.2.3: There are several possibilities:

```
succ1 = λn. λs. λz. s (n s z);
succ2 = λn. λs. λz. n s (s z);
succ3 = λn. plus c₁ n;
```

Solution to 4.2.6: Here's a simple one:

```
equal = λm. λn.
          and (iszro (m prd n))
              (iszro (n prd m));
```

Solution to 4.2.7: This is the solution I had in mind:

```
nil = λhh. λtt. tt;
cons = λh. λt. λhh. λtt. hh h (t hh tt);
head = λl. l (λh.λt.h) fls;
tail = λl.
         fst (l (λx. λp. pair (snd p) (cons x (snd p)))
                (pair nil nil));
isnull = λl. l (λh.λt.fls) tru;
```

Here is a rather different approach:

```
nil = pair tru tru;
cons = λh. λt. pair fls (pair h t);
head = λz. fst (snd z);
tail = λz. snd (snd z);
isnull = fst;
```

Solution to 4.2.8:

```
ff = λf. λl.
       test (isnull l)
          (λx. c₀) (λx. (plus (head l) (f (tail l)))) c₀;
sumlist = fixpoint ff;

l = cons c₂ (cons c₃ (cons c₄ nil));
equal (sumlist l) c₉;
```

▸ `λx. λy. x`

Solution to ??: For convenience, let's define a little notation first. If Γ is a naming context, write $\Gamma(\mathtt{x})$ for the index of `x` in Γ, counting from the right. Write $\Gamma \setminus \mathtt{x}$ for the context that results when `x` is removed from Γ. For example, if $\Gamma = \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}$, then $\Gamma(\mathtt{c}) = 1$ and $\Gamma \setminus \mathtt{c} = \mathtt{a}, \mathtt{b}, \mathtt{d}$.

Now, the property that we want is that, if $\mathtt{x} \notin FV(\mathtt{s})$, then

$$removenames_{\Gamma \setminus \mathtt{x}}(\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}) \quad = \quad \{\Gamma(\mathtt{x}) \mapsto removenames_{\Gamma \setminus \mathtt{x}}(\mathtt{s})\}(removenames_{\Gamma}(\mathtt{t})).$$

The proof proceeds by induction on `t`, using Definitions 4.3.2.2 and **??**, some simple calculations, and some easy lemmas about *removenames* and the other basic operations on terms. Convention 4.3.2.1 plays a crucial role in the abstraction case.

Solution to 5.3.3.3: Here's a counterexample: the term `(if false then true else 0)` is ill-typed, but reduces to the well-typed term `0`.

Solution to 6.2.3: T-TRUE and T-FALSE are introduction rules. T-IF is an elimination rule.

Solution to 6.3.1: Because the set of type expressions is empty (there is no base case in the syntax of types).

Solution to 6.4.2.1: The proof proceeds by induction on a derivation of $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathtt{T}$. There are cases for each of the typing rules (or, equivalently, each of the possible forms of `t`).

Case T-VAR*:* $\mathtt{t} = \mathtt{z}$
with $\Gamma(\mathtt{z}) = \mathtt{T}$

There are two sub-cases to consider, depending on whether `z` is the same as `x` or different. If $\mathtt{z} = \mathtt{x}$, then $\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{z} = \mathtt{s}$. The required result is then $\Gamma \vdash \mathtt{s} : \mathtt{S}$, which is among the assumptions of the lemma. Otherwise, $\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{z} = \mathtt{z}$, and the desired result is immediate.

Case T-ABS: $\quad$ $t = \lambda y{:}T_2.t_1$
$T = T_2 \to T_1$
$\Gamma, x{:}S, y{:}T_2 \vdash t_1 : T_1$

First note that, by convention 4.3.2.1, we may assume $x \neq y$ and $y \notin FV(s)$. Using weakening and permutation on the given subderivation, we obtain $\Gamma, y{:}T_2, x{:}S \vdash t_1 : T_1$. By the induction hypothesis, $\Gamma, y{:}T_2 \vdash \{x \mapsto s\}t_1 : T_1$. By T-ABS, $\Gamma \vdash \lambda y{:}T_2.\ \{x \mapsto s\}t_1 : T_2 \to T_1$. But this is the needed result, since, by the definition of substitution, $\{x \mapsto s\}t = \lambda y{:}T_1.\ \{x \mapsto s\}t_1$.

Case T-APP: $\quad$ $t = t_1\ t_2$
$\Gamma, x{:}S \vdash t_1 : T_2 \to T_1$
$\Gamma, x{:}S \vdash t_2 : T_2$
$T = T_2$

By the induction hypothesis, $\Gamma \vdash \{x \mapsto s\}t_1 : T_2 \to T_1$ and $\Gamma \vdash \{x \mapsto s\}t_2 : T_2$. By T-APP, $\Gamma \vdash \{x \mapsto s\}t_1\ \{x \mapsto s\}t_2 : T$, i.e., $\Gamma \vdash \{x \mapsto s\}(t_1\ t_2) : T$.

Case T-TRUE, T-FALSE: $\quad$ $t = 0$
$T = \mathtt{Bool}$

Then $\{x \mapsto s\}t = 0$, and the desired result, $\Gamma \vdash \{x \mapsto s\}t : T$, is immediate.

Case T-IF: $\quad$ $t = \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3$
$\Gamma, x{:}S \vdash t_1 : \mathtt{Bool}$
$\Gamma, x{:}S \vdash t_2 : T$
$\Gamma, x{:}S \vdash t_3 : T$

The induction hypothesis yields

$\Gamma \vdash \{x \mapsto s\}t_1 : \mathtt{Bool}$
$\Gamma \vdash \{x \mapsto s\}t_2 : T$
$\Gamma \vdash \{x \mapsto s\}t_3 : T$,

from which the result follows by T-IF. $\square$

Appendix B

Summary of Notation

B.1 Metavariable Conventions

In text	In ML code	Usage
p, q, r, s, t, u	s, t	terms
x, y, z	x, y	term variables
v, w	v, w	values
S, T, U, V	tyS, tyT	types
A, B, C	tyA, tyB	base types
$\alpha, \beta, \gamma, \delta$	alpha, beta	unification variables
X, Y, Z	tyX, tyY	type variables
J, K	kJ, kK	kinds
	bind	bindings
Γ, Δ	ctx	contexts
	fi	file position information
$\mathcal{J}$		arbitrary typing statements

B.2 Type System Names

Each type system discussed in the text can be identified by the symbol λ followed by a string of symbols representing the features that it includes. The following conventions prevent these strings from becoming too long and unwieldy:

1. If a component A extends a component B, then every type system including A will also include B; we then give just A in the name.

2. Also, since essentially every typed lambda-calculus includes the → type constructor, we elide it when naming more complex calculi.

For example, we write λF for the polymorphic lambda-calculus rathern than λ→XF.

Some combinations of typing features are sufficiently familiar to have established names in the literature:

SYSTEM	OTHER NAMES	
λ		untyped or pure lambda-calculus
$\lambda{\rightarrow}\mathtt{X}$	$\lambda^{\rightarrow}$, F_1	simply typed lambda-calculus
$\lambda{\rightarrow}\mathbb{N}\mu$	PCF	simple types with numbers and recursion

Appendix C

Running the Checkers

The systems discussed in the text have all been implemented in Objective CAML. You'll find the programming exercises and experiments throughout the notes much more enjoyable if you use the implementations to check and execute your solutions.

C.1 Preparing Your Input

An input file for a checker consists of a sequence of clauses terminated by semicolons. Input clauses can have the following forms:

`t;`	typecheck (if appropriate) and evaluate term `t`
`x = t;`	typecheck `t` (if appropriate) and bind it to `x`
`import "filename";`	include `filename` at this point

C.2 Ascii Equivalents

Programs in the text are typeset using some non-ascii symbols. When preparing input to the typechecker, you should use the following equivalents:

TYPESET	ASCII
λ	`lambda`
$\rightarrow$	`->`
$'$a	`'a`
$'$b	`'b`
$'$c	`'c`
$'$d	`'d`

C.3 Running the Checker

Precompiled binaries for solaris can be found in the course web directory. From CIS machines, you should be able to run the checkers directly out of the web directory:

```
/mnt/saul/extra/bcpierce/pub/courses/700/checkers/<checkername>/f <test.f>
```

(where `<checkername>` is the name of the checker that you want—`untyped`, `fulluntyped`, `simple`, etc.—and `<test.f>` is the name of your input file).

C.4 Compiling the Checkers

To modify the checker itself, you'll need to make a local copy:

```
cp -r /mnt/saul/extra/bcpierce/pub/courses/700/checkers/<checkername> <mydir>
```

To recompile it from scratch, first install the Objective CAML compiler if necessary (see the following section). Do

```
make clean
```

to throw away all the existing object-code files, and then

```
make
```

to rebuild the executable file (called `f`) for the checker.

C.5 Objective Caml

On CIS machines, you should be able to use the Objective CAML compiler simply by adding

```
/mnt/saul/extra/bcpierce/bin/sun4
```

to your search path.

On other machines (including most varieties of Unix workstations, Macs, and Windows 95/98/NT PCs) you'll need to install Objective Caml yourself. It can be obtained from `ftp://ftp.inria.fr/lang`. It's quite easy to install.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AS85] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, New York, 1985.

[Bac81] John Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[Bar84] H. P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[Bar90] H. P. Barendregt. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, volume B, chapter 7, pages 321–364. Elsevier / MIT Press, 1990.

[Bar92] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992.

[BCGS91] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[BDMN79] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[Ben95] Johan Van Benthem. *Language in Action : Categories, Lambdas, and Dynamic Logic*. MIT Press, 1995.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[BM92] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[Bru91] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[Car89] Felice Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Car96] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CCHO89] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, pages 457–467, 1989.

[CDC78] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.

[CDCS79] M. Coppo, M. Dezani-Ciancaglini, and P. Salle. Functional characterization of some semantic equalities inside λ-calculus. In Hermann A. Maurer, editor, *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, volume 71 of *LNCS*, pages 133–146, Graz, Austria, July 1979. Springer.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958. (Second edition, 1968).

[CG91] Pierre-Louis Curien and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 731–749. Springer-Verlag, September 1991.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CHO88] Peter Canning, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[Coq85] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, University Paris VII, January 1985.

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title *Multiple Inheritance via Intersection Types*.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw-Hill Book Co., New York, N.Y., 1992.

[Ghe90] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[Ghe95] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1,2):131–162, 1995.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, MA, 1992.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, Portland, OR, January 1994.

[HMV93] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25. IEEE Computer Society Press, 1993.

[How80] W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 303–310. ACM Press, January 1991. This paper presents the first algorithm for reconstructing the types and effects of expressions in the presence of first-class procedures in a polymorphically typed language. The algorithm involves a new technique called *algebraic reconstruction*, whose soundness and completeness properties are proved.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[LAB$^+$81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, February and March 1965.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

[LW91] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, FL*, pages 291–302. ACM, New York, NY, 1991.

[Mac86] David MacQueen. Using dependent types to express modular structure. In *13th Annual ACM Symposium on Principles of Programming languages*, pages 277–286, St. Petersburg Beach, FL, January 1986.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, Amsterdam, 1973. North Holland.

[Mar82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*. North Holland, Amsterdam, 1982.

[Mar88] Simone Martini. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[Mit84a] John C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.

[Mit84b] John C. Mitchell. Type inference and type containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 257–278, Berlin, June 1984. Springer LNCS 173. Full version in *Information and Computation*, vol. 76, no. 2/3, 1988, pp. 211–249. Reprinted in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990) 153–194.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, Amsterdam, 1990.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, 1996.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages*, 1985.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[N^+63] P. Naur et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6:1–17, January 1963.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 January 1997.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX.

[NL98] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. SV, 1998.

[oD80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Preliminary version in POPL '92.

[Pie97] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997. Summary version appeared in Conference on Typed Lambda Calculi and Applications, March 1993, pp. 346–360. Preliminary version available as University of Edinburgh technical report ECS-LFCS-92-200.

[PJL92] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

[Pot80] Garrell Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Preliminary version in Principles of Programming Languages (POPL), 1993.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Rey74] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[Rey80] John Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[SJ75] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass., December 1975.

[SNP90] Jan Smith, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL : A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 181–192, New York, May 21–24 1996. ACM Press.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

[vBM97] J. F. A. K. van Benthem and Alice Ter Meulen, editors. *Handbook of Logic and Language*. MIT Press, 1997.

[WAL+89] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[WR25] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica, Volume 1*. Cambridge University Press, Cambridge, 1925.

[Wri92] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

Index

$\longrightarrow^*$ (multi-step evaluation), 18
$\longrightarrow$ (one-step evaluation), 18, 34

abstract grammar, 14
abstract syntax tree, 17
Ad-hoc polymorphism, 88
algorithmic subtyping, 70

beta-reduction, 25
bound variable, 24

capture-avoiding substitution, 32
Church numerals, 27
Church-style, 43
closed term, 24
combinator, 24
Curry-style, 43
Currying, 25

depth of a term, 17
derivable, 38

elimination rule, 44
explicitly typed, 43
expression, *see* term

fixed-point combinator, 29
free variable, 24
free variables, 30

generation lemma, *see* inversion lemmas

higher-order functions, 25

identity function, 24
implicitly typed, 43
inductive definition, 14
introduction rule, 44

inversion lemmas
  typing, 39, 47

lambda-calculus, 22, 23
lambda-term, *see* term
lexicographic induction, 10

metalanguage, 24
metavariable, 24

natural-number induction, 10
normal form, 18, 29

object, 77
object calculus, 22
object language, 24
Overloading, 88

Parametric polymorphism, 88
pi-calculus, 22
preservation theorem, 39, 49
progress theorem, 40, 49
pure lambda-calculus, 23

rules
  EV-APP1, 34, 45, 58, 67, 90, 116
  EV-APP2, 34, 45, 58, 67, 90, 116
  EV-ASSIGN, 59
  EV-ASSIGN1, 59
  EV-ASSIGN2, 59
  EV-BETA, 25, 34, 45, 58, 67, 90, 116
  EV-BETA2, 90, 116
  EV-BETABOOLF, 18, 21, 46, 58
  EV-BETABOOLT, 18, 21, 46, 58
  EV-BETANATIS, 18, 21, 59
  EV-BETANATIZ, 18, 21, 59
  EV-BETANATPS, 18, 21, 59
  EV-BETANATPZ, 18, 21, 59

EV-COERCE1, 58
EV-DEREF, 59
EV-DEREF1, 59
EV-IF, 18, 21, 46, 58
EV-ISZERO, 18, 21, 59
EV-PACK, 104, 138
EV-PACKBETA, 104, 138
EV-PRED, 18, 21, 59
EV-PROJ, 53, 58
EV-RCDPROJ, 53, 58
EV-RECORD, 53, 58
EV-REF, 59
EV-REF1, 59
EV-SUCC, 18, 21, 59
EV-TAPP, 91, 116
EV-UNPACK, 104, 138
M-RCD, 54, 55
M-VAR, 54, 55
P-RCD, 56
P-VAR, 56
S-ALL, 117, 118, 132
S-ARROW, 65, 67, 117
S-EQV, 68
S-RCD, 69
S-RCD-DEPTH, 65, 68, 120
S-RCD-WIDTH, 65, 68, 120
S-REFL, 67, 117
S-REFL-TVAR, 129
S-SOME, 138
S-TOP, 66, 67, 117
S-TRANS, 65, 67, 117
S-TVAR, 115–117
SA-ALL, 130, 132
SA-ARROW, 70, 130, 132
SA-RCD, 70
SA-REFL-TVAR, 130, 132
SA-TOP, 70, 130, 132
SA-TRANS-TVAR, 130, 132
T-ABS, 43, 45, 61, 68, 91, 117
T-APP, 44, 45, 61, 64, 68, 91, 117
T-ASSIGN, 60
T-COERCE, 60
T-DEREF, 60
T-EQ, 63
T-FALSE, 38, 46, 60
T-FIX, 60
T-IF, 38, 46, 60
T-INERT, 60
T-ISZERO, 37, 61
T-LET, 53, 55, 60
T-LOC, 60
T-PACK, 102, 105, 138
T-PRED, 37, 61
T-PROJ, 54, 60
T-RCD, 54, 60
T-REF, 60
T-SUB, 64, 68, 117
T-SUCC, 37, 60
T-TABS, 90, 91, 117
T-TAPP, 90, 91, 117
T-TRUE, 38, 46, 60
T-UNIT, 52, 60
T-UNPACK, 102, 105, 138
T-VAR, 43, 45, 60, 68, 91, 117
T-ZERO, 37, 60
TA-ABS, 72, 127
TA-APP, 72, 127
TA-DOT, 72
TA-RCD, 72
TA-TABS, 127
TA-TAPP, 127
TA-VAR, 72, 127
TEQV-RCD-PERM, 63, 68
TEQV-REFL, 63, 68
TEQV-SYMM, 63, 68
TEQV-TRANS, 63, 68

simple types, 42
simply typed lambda-terms, 42
size of a term, 16
soundness, 39
stuck term, 19
subject expansion, 40
subject reduction, *see* preservationo
subtype polymorphism, 88

term, 14
Type systems and components
    (), 52–57, 63, 67, 68, 90, 104, 116, 118, 138
    Arithmetic expressions and Booleans ($+\mathbb{B}$), 21
    Simply typed lambda-calculus ($\lambda\to$), 45

Simply typed lambda-calculus with booleans ($\lambda{\rightarrow}\mathbb{B}$), 46
Untyped lambda-calculus (λ_u), 34
types, 37
typing assertions, 38
typing context, 43
typing derivation, 38
typing judgements, 38
typing relation, 38

untyped lambda-calculus, 23

value, 18
variable capture, 32

well-founded induction, 10
well-founded set, 10

Y combinator, 29