Type Systems
for Programming Languages

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

Working draft of February 24, 2000

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

Contents

Preface

The study of *type systems* for programming languages has emerged over the past decade as one of the most active areas of computer science research, with important applications in software engineering, programming language design, high-performance compiler implementation, and security of information networks.

This text aims to introduce the area to beginning graduate students and advanced undergraduates. A broad range of core topics are covered in detail, including simple type systems, type reconstruction, universal and existential polymorphism, subtyping, bounded quantification, recursive types, and type operators. Early chapters on the untyped lambda-calculus help make the book self-contained, allowing it to be used in courses for students with no background in the theory of programming languages.

The book adopts a strongly pragmatic approach throughout: a typical chapter begins with programming examples motivating a new typing feature, develops the feature and its basic metatheory, adduces typechecking algorithms, and illustrates these with executable ML typecheckers. The underlying semantic formalism is almost entirely operational, for a close correspondence with familiar programming languages.

A repeated theme is the theoretical properties required to build sound and complete typechecking algorithms. Both the proofs and the algorithms themselves are presented in detail. Moreover, each chapter is accompanied by a running implementation that can be used as a basis for experimentation by students, class projects, etc.

Audience

The book is aimed at graduate students, including both the general graduate population as well as students intending to specialize in programming language research. For the former, it should serve to introduce a number of key ideas from the design and analysis of programming languages, with type systems as an organizing structure. For the latter, it should provide sufficient background to proceed directly on to the research literature. The book is suitable for self-study by researchers in other areas who want to learn something about type systems, and should be a useful reference for experts. The first several chapters should be usable in upper-division undergraduate courses.

Goals

The main goals of the book are:

- Accessibility. A reader should be able to approach the book with little or no background in theory of programming languages.
- Coverage of core topics. By the end of the book, the reader should be fully equipped to tackle the research literature in type systems.
- Pragmatism. The book stays as close as possible to programming languages (sometimes at the expense of topics that might be included in a book written from the perspective of typed lambda-calculi and logic). In particular, the underlying computational substrate is a call-by-value lambda-calculus.
- Diversity. The book tries to show the leafiness of the topic rather than trying to unify all the threads addressed in the book. (Of course, I've unified as many of the threads as I could manage.)
- Honesty. Every type system discussed in the book is implemented. Every example runs.

To achieve all this, a few other desirable properties have been sacrificed.

- Completeness of coverage (probably impossible in one book, certainly in a textbook).
- Efficiency (as opposed to termination) of the typechecking algorithms described. This is not a book about industrial-strength typechecker implementation.

Required Background

No background in the theory of programming languages is assumed, but students should approach the book with a degree of prior mathematical maturity (in particular, rigorous undergraduate coursework in discrete mathematics, algorithms, and elementary logic).

Readers should also be familiar with some higher-order functional programming language (Scheme, ML, Haskell, etc.), and basic concepts of programming languages and compilers (abstract syntax, Backus-Naur grammars, evaluation, abstract machines, etc.). This material is available in many excellent undergraduate texts. I particularly like the one by Friedman, Wand, and Haynes [FWH92].

Course Outlines

In an advanced graduate course, it should be possible to cover essentially the whole book in a semester. For an undergraduate or beginning graduate course, there are two basic paths through the material:

1. type systems in programming (omitting implementation chapters), and

2. basic theory and implementation (omitting or skimming the end of the book).

Shorter courses can also be constructed by selecting particular chapters of interest. Figure 1 gives a sample syllabus from an introductory graduate-level course at Penn (two lectures a week of 90 minutes each).

In a course where term projects are a major part of the work, it may be desirable to postpone some of the theoretical material (e.g., denotational semantics, and perhaps some of the deeper chapters on implementation) so that a broad range of examples can be covered before the point where students have to choose project topics.

Chapter Dependencies

The major dependencies between chapters are outlined in Figure 2. Solid lines indicate that the later chapter is best read after the earlier. Dotted lines mean that the chapters can be read in any order except for some sections.

Typographic Conventions

Most chapters develop the features of some type system in a discursive way, then define the system formally as a collection of inference rules with brackets above and below to set them off from the surrounding text. For the sake of completeness, these definitions are usually presented in full, including not only the new rules for the features under discussion at the moment, but also the rest of the rules needed to constitute a complete calculus. The new parts are set on a gray background to make the "delta" from previous systems visually obvious.

An unusual feature of the book's production is that all the examples are mechanically checked during typesetting: an automatic script goes through each chapter, extracts the examples, generates and compiles a custom typechecker containing just the features under discussion, applies it to the examples, and inserts the checker's responses in the text. The system that does the hard parts of this, called *TinkerType*, was developed by Michael Levin and myself [LP99].

	Actual:	
1.	Course outline; history; administrivia	1
2.	Preliminaries: syntax and semantics	2 (outside class), 3
3.	Introduction to the lambda-calculus	4.1, 4.2
4.	Formalizing the lambda-calculus	4.3, 5
5.	Basics of types; the simply typed lambda calculus	6, 7
6.	Simple extensions; derived forms	9
7.	More extensions	9
8.	Normalization	8
9.	Exceptions; references	10, 11
	Tentative:	
10.	Equivalence; definitions	23, 24
11.	Subtyping	12
12.	Subtyping; imperative objects	14
13.	Recursive types	15
14.	Recursive types	15
15.	Structural vs. nominal presentations	16
16.	Case study: Featherweight Java	16
17.	Type reconstruction	17
18.	Universal polymorphism	18
19.	Existential polymorphism; ADTs	19
20.	Bounded quantification	20, 21
21.	Type operators	25
22.	Type operators (metatheory)	26
23.	Higher-order subtyping; polymorphic update	28, 29
24.	Purely functional objects	30
25.	Overflow	
26.	Overflow	

Figure 1: Sample syllabus

Figure 2: Chapter dependencies

Electronic resources

A collection of implementations for the typecheckers and interpreters described in the text is available at `http://www.cis.upenn.edu/~bcpierce/typesbook`. These implementations have been polished for readability and modifiability, and have been used successfully by my students as the basis of both small implementation exercises and larger course projects. The implementation language is the Objective Caml dialect of ML, freely available through `http://caml.inria.fr`.

Corrections for any errors discovered in the text will also be made available at `http://www.cis.upenn.edu/~bcpierce/typesbook`.

Acknowledgements

Many!

Chapter 1

Introduction

Proofs of programs are too boring for the social process of mathematics to work.
— Richard DeMillo, Richard Lipton, and Alan Perlis [DLP79]

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

Some writing remains in this chapter; what's here should be regarded as a detailed sketch. It's not worth making detailed comments at the level of sentences, since all that will change. Comments on the coherence of the ideas are very welcome.

Despite decades of concern in both industry and academia, expensive failures of large software projects are common. Proposed approaches to improving software quality include—among other ideas—a broad spectrum of techniques for helping ensure that a software system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior. On one end of this spectrum are powerful frameworks such as algebraic specification languages, modal logics, and denotational semantics; these can be used to express very general correctness properties but are cumbersome to use and demand significant involvement by the programmer not only in the application domain but also in the formal subtleties of the framework itself. At the other end are techniques of much more limited power—so limited that they can be built into compilers or linkers and thus "applied" even by programmers unfamiliar with the underlying theories. Such methods often take the form of *type systems*.

1.1 What is a Type System?

Despite the area's long history and huge research literature, it is quite difficult to define the term "type system" in a way that covers most informal usage but is still specific enough to have any bite. Here is my best attempt:

> *A type system is a clearly specified static method for proving the absence of certain program behaviors.*

Or, perhaps better:

> *A type system is a syntactic method for enforcing disciplined programming.*

Unpacking this definition a little...

- *program behaviors*: This book is concerned with type systems for programming languages. The term "type systems" (or, perhaps more commonly, "type theory") is also used in other parts of computer science and mathematics, where the focus is on connections between types and logical propositions (the "Curry-Howard isomorphism"). Similar concepts, notations, and techniques apply, but there are some important differences in orientation. For example, mathematical type theory is typically concerned with systems in which every well-typed computation is guaranteed to terminate, whereas most programming languages sacrifice this property in return for the convenience of features like recursive function definitions.

- *static*: (A type system is a *static* check—typically built into the compiler (and/or linker: cf. Java). From this perspective, terms like "dynamically typed" or "latently typed" are misnomers, better replaced by "dynamically checked.")

- *automatically*: (In programming languages, the use of the term "type system" is generally restricted to methods with efficient decision procedures. However, exactly what constitutes efficient is a matter of debate. Even widely used type systems like ML's may exhibit very poor (exponential) efficiency in pathological cases. Also, most type systems require some help from the programmer in terms of explicit annotations; at some point this annotation activity become full-blown theorem proving—where to draw the line is not at all clear. In practice, this is an area where terminology is driven by technology: when someone comes up with an analysis that can be performed "efficiently in practice" with "fairly light" programmer annotations, it tends to be called a type system. Projects like Compaq-SRC's Extended Static Checking play near the boundary.)

- *rigorously specified*: (A type system is usually considered to be part of the language—something that can be included in the language definition. A set of heuristics implemented by the compiler or another analysis tool is not usually called a type system, on this account. However, this distinction can become difficult (and probably not very productive). For example, methods like "soft typing" for Scheme lie on the boundary.)

- *absence*: (Being static, type systems are necessarily conservative: they can prove the absence of some bad behaviors, but they may sometimes reject programs in which, dynamically, these bad behaviors are never encountered. E.g. `if <complex calculation> then true(5) else 0`.)

- *certain*: (In practice, no type system can prevent every sort of bad dynamic behavior: conditions like divide by zero, array index out of bounds, nontermination, etc. are typically (though not always!) taken to fall outside the type system's purview.)

- *proving*: (A type system should provide a *guarantee* that the behaviors it prevents are really prevented, and this guarantee should be something we can establish once and for all for all programs in the language. Terms like "weakly typed" are euphamisms for "unsound" or "wrong." We need to be careful, however: some type systems are explicitly designed to defer certain checks until run-time for greater flexibility. Are these systems simply "unsafe," or should we (probably) consider them simply to be making a slightly weaker promise than they might first have appeared to be?)

  (Exactly which errors or nonsensical states a type system is intended to prevent is a matter to be decided for each language design: there is no single notion of "type safety." See the discussion below.)

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

1.2 What is a Type System For?

Efficiency

Safety

> *Well-typed programs cannot go wrong.*
>
> — Robin Milner [**?**]

The term "safety" is quite difficult to pin down. There is general agreement that some languages are safer than others, but each person's definition of precisely what this means is colored by the language framework that they prefer. My best shot:

> A safe language is one whose complete definition appears in the programmer's manual.

Every language comes with a language definition—i.e., the set of things you need to know to be able to predict the behavior of any program in the language. Sometimes (e.g. Scheme, ML, Haskell, Java) the language definition is the same as the language manual, i.e. the definition is written out somewhere very precisely, or even (SML) formally. For other languages (e.g. C, C++, Perl) some rules are written out more or less exactly, while others must be figured out or learned by experience — the "definition," in effect, is just the source code for a compiler or interpreter. Phrases like "if X happens, the results are undefined" are a shorthand for "to predict the result of X, you have to understand the compiler [and the runtime system, OS, and machine architecture]."

(Assembly languages can be regarded as safe in this sense (the "definition" is the machine architecture manual), but this is not a very interesting claim)

Implementation features like GC and aggressive optimization are much easier to implement in safe languages.

"Strongly typed" is often used as a synonym for "safe."

At this level, the erroneous behaviors are generally fairly simple, local things like branching to a float. Type systems are good at preventing many, but not all, such errors.

Safety can be achieved by both static and dynamic means (typically a combination). Some advantages of doing it statically when possible are:

- efficiency: a check can be done once at compile (or link) time instead of many times at run time
- early error detection

Of course, there are costs too:

- (possibly) heavier / more verbose syntax
- some safe programs will inevitably be excluded

Almost all general purpose languages – even seamless and civilized ones – offer programmers "escape hatches" (like calls to C) and warn them to use them with care because they break the safety properties. Indeed, such escape hatches are often provided within the language itself. OCaml has "magic." SML/NJ has `Unsafe.cast` (check!). M3 has `unsafe` modules that are explicitly permitted to perform operations whose correctness depends on a detailed understanding of the compiler and runtime system. Such features do weaken the claim that "well typed programs cannot go wrong," since they make it easy to write programs that dump core, scribble over arbitrary memory locations, etc. But at least, when the program does dump core, one knows that only the `unsafe` modules (and the compiler itself!) need to be examined.

A slightly different way of talking about safety: Cardelli [**?**] distinguishes between what he calls *trapped* and *untrapped errors*. Trapped errors cause a computation to stop immediately, while untrapped errors may allow the computation to continue (at least for a while). An example of an untrapped error might be accessing data beyond the end of an array in a language like C. Cardelli argues that a *safe* language is one that does not allow any untrapped errors to occur at run-time.

An oft-remarked (and somewhat puzzling) fact: This kind of checking exposes not only trivial mental slips, but also deeper conceptual errors, which frequently manifest as type errors. Programs that pass the typechecker tend to "just work" more often than you feel you have a right to expect. This effect depends on the type of problem you're working on and the degree of refinement of the type system, of course.

Documentation

Types are "checked documentation"

Support for Programmer-Defined Abstractions

> *Type structure is a syntactic discipline for enforcing levels of abstraction.*
> — John Reynolds [Rey83]

By "security" here I mean "language support for programmer-defined abstractions" [the word is not ideal — suggestions?] (Of course, only safe languages can be truly secure, but many unsafe languages provide approximations: the concepts are somewhat orthogonal—i.e., you can say "safe programs are secure," even though you can't decide mechanically which programs are safe.)

Here the erroneous behaviors are extended to include breaking programmer-defined abstractions. This form of security is a crucial requirement for large-scale programming.

type systems are a very important mechanism for achieving language security, but there are others

- procedural abstraction (simple objects)

- run-time support for generative tags (MzScheme)
- sandboxing, on-the-fly rewriting, and other run-time techniques
- OS-level mechanisms (separate address spaces)

Language Design

Type system design goes hand in hand with language design, and a typing discipline comes with a programming discipline.

Type systems should be part of the PL design: (Morrisett) The design of this notation for expressing invariants is a language design that requires a lot of careful thought. If it's not designed as a language, then programmers have no portable way of expressing their intention or interfacing to the analysis tools. In addition, when integrated into the language (or at least environment), a compiler can take advantage of the invariants. And finally, the ability to explicitly state invariants is crucial for giving understandable feedback to the programmer when the analysis fails.

(Development of the type system should go hand-in-hand with the language and that getting good results is as much a function of getting programmers to meet the type-checker in the middle as it is for sophisticated analyses. — Morrisett)

A statically typed language is not the same as an explicitly typed one. (Harper: A related point is the apparent inability of some writers to imagine type inference, equating typed languages with those that require huge amounts of type information to be explicitly and tediously maintained by the programmer. Ousterhout comes to mind here.)

1.3 History

The following table presents a (rough and incomplete) chronology of some important high points in the history of type systems in computer science. Related developments in logic are also included (in italics), to give a sense of the importance of this field's contributions.

late 1800s	*Origins of formal logic*	[?]
early 1900s	*Formalization of mathematics*	[WR25]
1930s	*Untyped lambda-calculus*	[Chu41]
1940s	*Simply typed lambda-calculus*	[Chu40, CF58]
1950s	Fortran	[Bac81]
1950s	Algol	[N^+63]
1960s	*Automath project*	[dB80]
1960s	Simula	[BDMN79]
1970s	*Martin-Löf type theory*	[Mar73, Mar82, SNP90]

1960s	*Curry-Howard isomorphism*	[How80]
1970s	*System F*, F $^{\omega}$	[Gir72]
1970s	polymorphic lambda-calculus	[Rey74]
1970s	CLU	[LAB^{+}81]
1970s	polymorphic type inference	[Mil78, DM82]
1970s	ML	[GMW79]
1970s	*intersection types*	[CDC78, CDCS79, Pot80]
1980s	NuPRL project	[Con86]
1980s	subtyping	[Rey80, Car84, Mit84a]
1980s	ADTs as existential types	[MP88]
1980s	*calculus of constructions*	[Coq85, CH88]
1980s	*linear logic*	[Gir87, GLT89]
1980s	bounded quantification	[CW85, CG92, CMMS94]
1980s	*Edinburgh Logical Framework*	[HHP92]
1980s	Forsythe	[Rey88]
1980s	*pure type systems*	[Bar92a]
1980s	dependent types and modularity	[Mac86]
1980s	Quest	[Car91]
1980s	*Extended Calculus of Constructions*	[Luo90]
1980s	Effect systems	[**?**, TJ92, TT97]
1980s	row variables and extensible records	[Wan87, Rém89, CM91]
1990s	higher-order subtyping	[Car90, CL91, PT94]
1990s	typed intermediate languages	[TMC^{+}96]
1990s	Object Calculus	[AC96]
1990s	translucent types and modarity	[HL94, Ler94]
1990s	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 integer-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 1950s and early 1960s (e.g., ALGOL), the classification was extended to structured data (arrays of records, etc.) and higher-order functions. Beginning in the 1970s, 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 Application Areas

Beyond their traditional benefits of robustness and efficiency, type systems play an increasingly central role in computer and network security: static typing lies at the core of the security models of Java and JINI, for example, and is the main enabling technology for Proof-Carrying Code. Type systems are used to organize compilers, verify protocols, structure information on the web, and even model natural languages.

Short sketches of some of these diverse applications...

- *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 (cf. Walker, Leroy and Rouaix, etc., etc)*
- *In theorem proving*
- *In databases*
- *In linguistics (categorial grammar [Ben95, vBM97, etc.] , and maybe something seminal by Lambek)*
- *In Y2K conversion tools*
- *DTDs and other "web metadata" (note from Henry Thompson: DTDs were originally designed for SGML because of the expense of cancelling huge typesetting runs due to errors in the markup!)*

1.5 Related Reading

While this book attempts to be self contained, it is far from comprehensive: the area is too large, and can be approached from too many angles, to do it justice in one book. Here are a few other good entry points:

- Handbook articles by Cardelli [Car96] and Mitchell [Mit90] offer quick introductions to the area. Barendregt's article [Bar92b] is for the more mathematically inclined.
- Mitchell's massive textbook on programming languages [Mit96] covers basic lambda calculus, a range of type systems, and many aspects of semantics.
- Abadi and Cardelli's *A Theory of Objects* [AC96] develops much of the same material as this present book, de-emphasizing implementation aspects and concentrating instead on the application of these ideas in a foundation treatment of object-oriented programming. Kim Bruce's forthcoming *Foundations of Object-Oriented Programming Languages* will cover similar ground. Introductory material on object-oriented type systems can also be found in [PS94, Cas97].
- Reynolds [Rey98] *Theories of Programming Languages*, a graduate-level survey of the theory of programming languages, includes beautiful expositions of polymorphic typing and intersection types.
- Girard's *Proofs and Types* [GLT89] treats logical aspects of type systems (the Curry-Howard isomorphism, etc.) thoroughly. It also includes a description of System F from its creator, and an appendix introducing linear logic.
- *The Structure of Typed Programming Languages*, by Schmidt [**?**], develops core concepts of type systems in the context of programming language design, including several chapters on conventional imperative languages. Simon Thompson's *Type Theory and Functional Programming* [Tho91] focuses on connections between functional programming (in the "pure functional programming" sense of Haskell or Miranda) and constructive type theory, viewed from a logical perspective.
- Semantic foundations for both untyped and typed languages are covered in depth in textbooks by Gunter [Gun92] and Winskel [Win93].
- Hindley's monograph *Basic Simple Type Theory* [Hin97] is a wonderful compendium of results about the simply typed lambda-calculus and closely related systems. Its coverage is deep rather than broad.

If you want a single book besides the one you're holding, I'd recommend either Mitchell or Abadi and Cardelli.

Chapter 2

Mathematical Preliminaries

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

2.1 Sets, Relations, and Sequences

2.1.1 Definition: We use standard notation for sets: curly braces for listing the elements of a set explicitly ($\{\ldots\}$) or showing how to construct one set from another by "comprehension" ($\{x \in \mathcal{S} \mid \ldots\}$), $\emptyset$ for the empty set, and $\mathcal{S} \setminus \mathcal{T}$ for the set difference of $\mathcal{S}$ and $\mathcal{T}$ (the set of elements of $\mathcal{S}$ that are not also elements of $\mathcal{T}$). □

2.1.2 Definition: An n-place **relation** between a collection of sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ is a set $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$ of tuples of elements from $\mathcal{S}_1$ through $\mathcal{S}_n$. We say that the elements $s_1 \in \mathcal{S}_1$ through $s_n \in \mathcal{S}_n$ are **related by** R if the tuple $(s_1, \ldots, s_n)$ is an element of R. □

2.1.3 Definition: A one-place relation on a set $\mathcal{S}$ is called a **predicate** on $\mathcal{S}$. We say that P is true of an element $s \in \mathcal{S}$ if $s \in P$. To emphasize this intuition, we often write $P(s)$ instead of $s \in P$, regarding P as a function mapping elements of $\mathcal{S}$ to truth values. □

2.1.4 Definition: A two-place relation R between sets $\mathcal{S}$ and $\mathcal{T}$ is called a **binary relation**. We often write $s \; R \; t$ instead of $(s, t) \in R$. When $\mathcal{S}$ and $\mathcal{T}$ are actually the same set $\mathcal{U}$, we say that R is a binary relation **on** $\mathcal{U}$. □

2.1.5 Definition: A relation R on a set $\mathcal{S}$ is **reflexive** if R relates every element of $\mathcal{S}$ to itself—that is, $s \; R \; s$ (or $(s, s) \in R$) for all $s \in \mathcal{S}$. R is **symmetric** if $s \; R \; t$ implies $t \; R \; s$, for all s and t in $\mathcal{S}$. R is **transitive** if $s \; R \; t$ and $t \; R \; u$ together imply $s \; R \; u$. R is **antisymmetric** if $s \; R \; t$ and $t \; R \; s$ together imply that $s = t$. □

2.1.6 Definition: A reflexive, transitive, and antisymmetric relation R on a set $\mathcal{S}$ is called a **partial order** on $\mathcal{S}$. (When we speak of "a partially ordered set S," we always have in mind some particular partial order R on $\mathcal{S}$.) Partial orders are usually written using symbols like $\leq$ or $\sqsubseteq$.

A partial order $\leq$ is called a **total order** if it also has the property that, for each s and t in $\mathcal{S}$, either $s \leq t$ or $t \leq s$. □

2.1.7 Definition: A **strict partial order** is an irreflexive, antisymmetric, and transitive relation on a set $\mathcal{S}$. Each partial order $\leq$ gives rise to a strict partial order $<$ defined in the obvious way: $s < t$ iff $s \leq t$ and $s \neq t$. □

2.1.8 Definition: A reflexive, transitive, and symmetric relation on a set $\mathcal{S}$ is called an **equivalence** on $\mathcal{S}$. □

2.1.9 Definition: Suppose R is a relation on a set $\mathcal{S}$. The **reflexive closure** of R is the smallest reflexive relation R' that contains R. ("Smallest" in the sense that if R'' is some other reflexive relation that contains all the pairs in R, then we have $\mathsf{R}' \subseteq \mathsf{R}''$.) Similarly, the **transitive closure** of R is the smallest transitive relation R' that contains R. The transitive closure of R is often written R^+. The **reflexive and transitive closure** of R is the smallest reflexive and transitive relation R' that contains R. It is often written R^*. □

2.1.10 Exercise: Suppose we are given a partial order R on a set $\mathcal{S}$. Define the relation R' as follows:

$$\mathsf{R}' = \mathsf{R} \cup \{(s, s) \mid s \in \mathcal{S}\}.$$

That is, R' contains all the pairs in R plus all pairs of the form (s, s). Show that R' is the reflexive closure of R. □

2.1.11 Definition: Suppose R is a relation on a set $\mathcal{S}$, and that P is a predicate on $\mathcal{S}$. We say that P is **preserved by** R if whenever we have $s \mathrel{\mathsf{R}} s'$ and $\mathsf{P}(s)$, we also have $\mathsf{P}(s')$. □

2.1.12 Definition: An **ordered sequence** is written by listing its elements, separated by commas. We use comma as both the "cons" operation for adding an element to either end of a sequence and as the "append" operation on sequences. For example, if a is the sequence $3, 2, 1$ and b is the sequence $5, 6$, then $0, a$ denotes the sequence $0, 3, 2, 1$, while $a, 0$ denotes $3, 2, 1, 0$ and b, a denotes $5, 6, 3, 2, 1$. The sequence of numbers from 1 to n is abbreviated $1..n$. We write $|a|$ for the length of the sequence a. (The same notation $|\mathcal{S}|$ is also used for the size of a set $\mathcal{S}$.) □

2.2 Induction

2.2.1 Definition: Suppose we have a partial ordering $\leq$ on a set $\mathcal{S}$. A **decreasing chain** in $\leq$ is a sequence $s_1, s_2, s_3, \ldots$ of elements of $\mathcal{S}$ such that each member of the sequence is strictly less than its predecessor: $s_{i+1} < s_i$ for every i. Chains can be either finite or infinite, but we are more interested in infinite ones, as in the next definition. □

2.2.2 Definition: Suppose we have a set $\mathcal{S}$ with a partial order $\leq$. The ordering $\leq$ is said to be **well founded** if it contains no infinite decreasing chains. For example, the usual ordering on the natural numbers, with $0 \leq 1 \leq 2 \leq 3 \leq \ldots$ is well founded, but the same ordering on the integers, $\ldots \leq -3 \leq -2 \leq -1 \leq 0 \leq 1 \leq 2 \leq 3 \leq \ldots$ is not. □

2.2.3 Axiom [Principle of well-founded induction]: Suppose we are given a well-founded ordering $\leq$ on a set $\mathcal{S}$ and a predicate P on $\mathcal{S}$. If we can show, for each $s \in \mathcal{S}$, that $(\forall s' < s.\ P(s'))$ implies $P(s)$, then we may conclude that $P(t)$ holds for every $t \in \mathcal{S}$. Graphically:

$$\frac{\forall s. \quad (\forall s' < s.\ P(s')) \Longrightarrow P(s)}{\forall t.\ P(t)}$$

This **inference rule** notation will be used heavily in what follows. Informally, it should be read "if all the conditions listed above the line are satisfied, we may infer that the statement below the line also holds." □

2.2.4 Corollary [Principle of complete 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. Graphically:

$$\frac{\forall m. \quad (\forall i < m.\ P(i)) \Longrightarrow P(m)}{\forall n.\ P(n)}$$

□

Proof: The usual ordering $\leq$ on the natural numbers is well founded. □

2.2.5 Definition: The **lexicographic ordering** (or "dictionary ordering") on pairs of natural numbers is defined as follows: $(m, n) \leq (m', n')$ iff either $m < m'$ or else $m = m'$ and $n \leq n'$. It is easy to check that the lexicographic ordering on pairs of natural numbers is well founded. This leads to another corollary of the principle of well-founded induction. □

2.2.6 Corollary [Principle of lexicographic induction]: 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). □

2.2.7 Exercise: Here is a more constructive definition of the transitive closure of a relation R. First, we define the following sequence of sets of pairs:

$$\begin{array}{lcl} R_0 & = & R \\ R_{i+1} & = & R_i \cup \{(s,u) \mid \text{for some } t, (s,t) \in R_i \text{ and } (t,u) \in R_i\} \end{array}$$

That is, we construct each R_{i+1} by adding to R_i all the pairs that can be obtained by "one step of transitivity" from pairs already in R_i. Finally, define the relation R^* as the union of all the R_i:

$$R^* = \bigcup_i R_i$$

Show that this R^* is really the transitive closure of R—i.e., that it satisfies the conditions given in Definition 2.1.9. □

2.2.8 Exercise: Suppose R is a relation on a set $\mathcal{S}$, and that P is a predicate on $\mathcal{S}$ that is preserved by R. Show that P is also preserved by R^*. □

2.3 Background Reading

If the material in this chapter is unfamiliar, you may want to start with some background reading. There are many sources for this, but Winskel's book [Win93] is a particularly good choice for intuitions about induction. The beginning of Davey and Priestley [**?**] has an excellent review of ordered sets.

Chapter 3

Untyped Arithmetic Expressions

3.1 Basics

We begin with a very simple language for calculating with numbers and booleans. This language provides the boolean constants `true` and `false`...

```
   true;
▸ true
```

conditional expressions...

```
   if false then true else false;
▸ false
```

the numeric constant `0`...

```
   0;
▸ 0
```

the arithmetic operators `succ` (successor) and `pred` (predecessor)...

```
   succ (succ (succ 0));
▸ 3

   pred (succ (succ 0));
▸ 1
```

and a testing operation `iszero` that returns `true` when it is applied to `0` and `false` when it is applied to some other number:

```
   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.

3.2 Formalities

Syntax

There are several equivalent ways of defining the syntax of our language.

3.2.1 Definition [Terms, informally]: 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 $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$. The set of terms is written $\mathcal{T}$. □

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

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

1. $\{\mathtt{true}, \mathtt{false}, 0\} \subseteq \mathcal{T}$;
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 $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \in \mathcal{T}$. □

Definition 3.2.2 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.

3.2.3 Definition [Terms, concretely]: For each natural number $\mathtt{i}$, define a set $\mathcal{S}_\mathtt{i}$ as follows:

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

Finally, let

$$\mathcal{S} \quad = \quad \bigcup_{i} \mathcal{S}_i.$$

$\mathcal{S}_0$ is empty; $\mathcal{S}_1$ contains just the constants; $\mathcal{S}_2$ 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 and 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 arithmetic and conditional operators, beginning with just constants. □

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

3.2.5 Exercise [Quick check]: Show that the sets $\mathcal{S}_i$ are **cumulative**—that is, that for each i we have $\mathcal{S}_i \subseteq \mathcal{S}_{i+1}$. □

Let us check that the two formal 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.

3.2.6 Proposition: $\mathcal{T} = \mathcal{S}$. □

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

For part (b), suppose that some set $\mathcal{S}'$ satisfies the three conditions in Definition 3.2.2. We will argue, by complete induction on i, that every $\mathcal{S}_i \subseteq \mathcal{S}'$, from which it clearly follows that $\mathcal{S} \subseteq \mathcal{S}'$.

Suppose that $\mathcal{S}_j \subseteq \mathcal{S}'$ for all $j < i$; we must show that $\mathcal{S}_i \subseteq \mathcal{S}'$. Since the definition of $\mathcal{S}_i$ has two clauses (for $i = 0$ and $i > 0$), there are two cases to consider.

- If $i = 0$, then $\mathcal{S}_i = \emptyset$. But $\emptyset \subseteq \mathcal{S}'$ trivially.
- Otherwise, $i = j + 1$ for some j. Let $\texttt{t}$ be some element of $\mathcal{S}_{j+1}$. Since $\mathcal{S}_{j+1}$ is defined as the union of three smaller sets, $\texttt{t}$ must come from one of these sets; there are three possibilities to consider:

    1. $\texttt{t}$ is a constant, hence $\texttt{t} \in \mathcal{S}'$ by condition (1).

2. `t` has the form `succ` t_1, `pred` t_1, or `iszero` t_1, for some $t_1 \in \mathcal{S}_j$. But then, by the induction hypothesis, $t_1 \in \mathcal{S}'$, and so, by condition (2), $t \in \mathcal{S}'$.
3. `t` has the form `if` t_1 `then` t_2 `else` t_3, for some $t_1, t_2, t_3 \in \mathcal{S}_i$. Again, by the induction hypothesis, t_1, t_2, and t_3 are all in $\mathcal{S}'$, and hence, by condition (3), so is `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. □

3.2.7 Definition [Terms, compactly]: The definitions we have seen so far have all been rather verbose. Programming language designers have invented a number of more concise and readable notations, mostly based on Backus-Naur Form grammars [**?**]. The variant of BNF used in the remainder of this book looks like this:

`t` ::=		*terms*
	`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*

The first line (`t` ::=) declares that we are defining the set of terms, and that we are going to use the letter `t` to range over terms. Each line that follows gives one alternative syntactic form for terms. At every point where a `t` appears, we may substitute any term. □

Abstract vs. Concrete Syntax

The preceding definition is called an **abstract grammar** because it ignores issues such as placement of parentheses—the fact that, for example, `succ (succ 0)` requires parentheses around `(succ 0)` whereas the `0` in `succ 0` does not require parentheses. In effect, the definition is really talking about **abstract syntax trees**, not about concrete strings of symbols or characters. We will ignore issues of concrete syntax, lexical analysis, parsing, etc. throughout the book.

Structural Induction

The explicit characterization of the set of terms $\mathcal{T}$ in Proposition 3.2.6 justifies an important principle for reasoning about its elements. If $\mathtt{t} \in \mathcal{T}$, then one of three things must be true about `t`—either

1. `t` is a constant, or
2. `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. `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 `t` to the set of constants used in `t`.

3.2.8 Definition: The set of constants appearing in a term `t`, written *Consts*(`t`), is defined as follows:

$$
\begin{array}{lcl}
\mathit{Consts}(\mathtt{true}) & = & \{\mathtt{true}\} \\
\mathit{Consts}(\mathtt{false}) & = & \{\mathtt{false}\} \\
\mathit{Consts}(\mathtt{0}) & = & \{\mathtt{0}\} \\
\mathit{Consts}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{Consts}(\mathtt{t}_1) \\
\mathit{Consts}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \cup \mathit{Consts}(\mathtt{t}_3)
\end{array}
\qquad \square
$$

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

3.2.9 Definition: The **size** of a term `t`, written *size*(`t`), is defined as follows:

$$
\begin{array}{lcl}
\mathit{size}(\mathtt{true}) & = & 1 \\
\mathit{size}(\mathtt{false}) & = & 1 \\
\mathit{size}(\mathtt{0}) & = & 1 \\
\mathit{size}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{size}(\mathtt{t}_1) + 1 \\
\mathit{size}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) + 1
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{depth}(\mathtt{true}) & = & 1 \\
\mathit{depth}(\mathtt{false}) & = & 1 \\
\mathit{depth}(\mathtt{0}) & = & 1 \\
\mathit{depth}(\mathtt{succ}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{pred}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{iszero}\ \mathtt{t}_1) & = & \mathit{depth}(\mathtt{t}_1) + 1 \\
\mathit{depth}(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & \max(\mathit{depth}(\mathtt{t}_1), \mathit{depth}(\mathtt{t}_2), \mathit{depth}(\mathtt{t}_3)) + 1
\end{array}
$$

Equivalently, $\mathit{depth}(\mathtt{t})$, is the smallest i such that $\mathtt{t} \in \mathcal{S}_i$ according to Definition 3.2.3. □

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

3.2.10 Lemma: The number of distinct constants in a term $\mathtt{t}$ is no greater than the size of $\mathtt{t}$ ($|\mathit{Consts}(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$). □

Proof: The 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.

The proof proceeds by induction on the depth of $\mathtt{t}$. That is, assuming the desired property for all terms "shorter 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: $|\mathit{Consts}(\mathtt{t})| = |\{\mathtt{t}\}| = 1 = \mathit{size}(\mathtt{t})$.

Case: $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1, \mathtt{pred}\ \mathtt{t}_1,$ or $\mathtt{iszero}\ \mathtt{t}_1$

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

Case: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$

By the induction hypothesis, $|\mathit{Consts}(\mathtt{t}_1)| \leq \mathit{size}(\mathtt{t}_1)$ and $|\mathit{Consts}(\mathtt{t}_2)| \leq \mathit{size}(\mathtt{t}_2)$ and $|\mathit{Consts}(\mathtt{t}_3)| \leq \mathit{size}(\mathtt{t}_3)$. We now calculate as follows: $|\mathit{Consts}(\mathtt{t})| = |\mathit{Consts}(\mathtt{t}_1) \cup \mathit{Consts}(\mathtt{t}_2) \cup \mathit{Consts}(\mathtt{t}_3)| \leq |\mathit{Consts}(\mathtt{t}_1)| + |\mathit{Consts}(\mathtt{t}_2)| + |\mathit{Consts}(\mathtt{t}_3)| \leq \mathit{size}(\mathtt{t}_1) + \mathit{size}(\mathtt{t}_2) + \mathit{size}(\mathtt{t}_3) < \mathit{size}(\mathtt{t})$. □

The form of this proof can be clarified by restating it as a general reasoning principle. Compare this principle with the complete induction principle for natural numbers on p. 24.

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

$$\frac{\forall \mathtt{s}. \quad (\forall \mathtt{r} \text{ with } \mathit{depth}(\mathtt{r}) < \mathit{depth}(\mathtt{s}).\ \mathsf{P}(\mathtt{r})) \Longrightarrow \mathsf{P}(\mathtt{s})}{\forall \mathtt{s}.\ \mathsf{P}(\mathtt{s})}$$

□

Proof: Exercise. □

Note that almost nothing changes if we replace *depth* with *size* in the above theorem.

Another common induction principle on terms is closely analogous to the principle of ordinary mathematical induction on the natural numbers:

$$\frac{\forall \mathtt{s}. \quad (\forall \text{ immediate subterms } \mathtt{r} \text{ of } \mathtt{s}.\ \mathsf{P}(\mathtt{r})) \Longrightarrow \mathsf{P}(\mathtt{s})}{\forall \mathtt{s}.\ \mathsf{P}(\mathtt{s})}$$

For simple proofs, it often makes little difference whether we argue by induction on the size, depth, or structure of terms.

Evaluation

Explain:

- *Some decisions to be made about details: e.g. predecessor of `0` (we'll choose it to be `0`) and `succ false` (we'll consider it an error)*
- *Operational vs. denotational semantics*
- *Plotkin's SOS*
- *Evaluation contexts vs. congruence rules*
- *How evaluation order is specified by the use of restricted metavariables like `v` and `nv`*
- *might want to point out there that just because `t` $\rightarrow$ `t'` it doesn't necessarily follow that `T[t]` $\rightarrow$ `T[t']` where `T[.]` is a term with a hole in it*

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

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

3.2.13 Definition: The **one-step evaluation** relation $\longrightarrow$ is the smallest binary relation on terms containing all instances of the following rules:

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

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

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

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

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-BetaNatPZ)}$$

$$\texttt{pred (succ nv)} \longrightarrow \texttt{nv} \qquad \text{(E-BetaNatPS)}$$

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

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-BetaNatIZ)}$$

$$\texttt{iszero (succ nv)} \longrightarrow \texttt{false} \qquad \text{(E-BetaNatIS)}$$

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

□

Explain:

- *inference rules (premises and conclusion, etc.)*
- *role of metavariables (rules vs. rule schemes)*
- *derivation trees (show derivation trees for a couple of evaluation derivations and ask them to draw one or two more)*
- *inductive proofs over derivation trees*
- *computation rules vs. congruence rules*

3.2.14 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 $\texttt{t} \longrightarrow^* \texttt{t}''$. □

3.2.15 Exercise [Quick check]: Rewrite the previous definition using a set of inference rules to define the relation $\mathtt{t} \longrightarrow^* \mathtt{t}'$. Solution on page 280. □

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

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

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

Properties of Evaluation

3.2.18 Proposition: Every value is in normal form. □

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

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

Proof: Exercise. Solution on page 280. □

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

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

Proof: Exercise. □

Run-Time Errors

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

"Stuckness" gives us a simple notion of "run-time 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 different way of formalizing these meaningless states is to introduce a new term called wrong and augment the operational semantics with rules that generate wrong in all the situations where the old semantics would get stuck.

3.2.23 Definition: First, we add a new constant wrong to the set of terms. Next, we introduce the following new syntactic categories:

badnat ::=	*normal forms that are not numbers*
wrong	*runtime error*

```
        true                                    constant true
        false                                   constant false
badbool ::=                                     normal forms that are not booleans
        wrong                                   runtime error
        nv                                      numeric value
```

Finally, we add the following rules to the evaluation relation:

$$\texttt{if badbool then } \texttt{t}_1 \texttt{ else } \texttt{t}_2 \longrightarrow \texttt{wrong} \qquad \text{(E-IF-WRONG)}$$

$$\texttt{succ badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-SUCC-WRONG)}$$

$$\texttt{pred badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-PRED-WRONG)}$$

$$\texttt{iszero badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-ISZERO-WRONG)}$$

□

3.2.24 Exercise [Homework]: *(Due on Feb. 3.)* Prove that these two treatments of run-time errors agree in a suitable sense. (Hint: As is often the case when proving things about programming languages, the only tricky part about the exercise is finding the right way of formulating a precise statement to be proved—the proof itself is fairly straightforward.) Solution on page 281. □ ⇐

3.3 A Concrete Implementation

Working with formal definitions such as those in the previous section is often easier when the intuitions behind the definitions are "grounded" by a connection to a concrete implementation. We sketch here the key components of an implementation of our language of booleans and arithmetic expressions.

The code presented here (and in the implementation sections throughout the book) is written in Objective Caml [**?**], a popular dialect of ML [**?**]. Only a small subset of the full OCaml language is used; it should be easy to translate the examples here into other languages that the reader may prefer. The most important requirements are automatic storage management and easy facilities for defining recursive functions by pattern matching over structured data types. Other functional languages such as Standard ML [**?**], Haskell [**?**], and Scheme [**?**] are fine choices. Languages with garbage collection but without pattern matching, such as Java, are rather heavy for the sorts of programming we'll be doing. Languages with neither, such as C, are even less suitable.

Syntax

Our first job is to define a type of OCaml values representing terms. OCaml's datatype definition mechanism makes this easy: the following declaration is a straightforward transliteration of Definition 3.2.7.

```
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 constructors `TmTrue` to `TmIsZero` name the different sorts of nodes in the abstract syntax trees of type `term`; the type following `of` in each case specifies the number of subtrees that will be attached to that type of node.

Each abstract syntax tree node is annotated with a value of type `info`, which describes what character position in the source file where the node originated. This information is created by the parser when it scans the input file, and it is used by printing functions to indicate to the user where an error occurred. For purposes of understanding the basic algorithms of evaluation, typechecking, etc., this information could just as well be omitted; it is included here only so that readers who wish to experiment with the implementations themselves will see the code in exactly the same form as discussed in the book.

In the definition of the evaluation relation, we'll need to check whether a term is a numeric value:

```
let rec isnumericval = function
    TmZero(_) → true
  | TmSucc(_,t0) → isnumericval t0
  | _ → false
```

This is a typical example of recursive definition by pattern matching in OCaml: `isnumericval` is defined as the function that, when applied to `TmZero`, returns true; when applied to `TmSucc` with subtree `t0` makes a recursive call to check whether `t0` is a numeric value; and when applied to any other term returns `false`. The function that checks whether a term is a value is similar:

```
let rec isval = function
    TmTrue(_)  → true
  | TmFalse(_) → true
  | t when isnumericval t  → true
  | _ → false
```

The third clause is a "conditional pattern": it matches any term `t`, but only so long as the boolean expression `isnumericval t` yields `true`.

Evaluation

The implementation of the evaluation relation closely follows the single-step evaluation rules in Definition 3.2.13. As we have seen, these rules define a *partial* function that, when applied to a term that is not yet a value, yields the next step in the evaluation sequence for that term. When applied to a value, the result of the evaluation function yields no result. To translate the evaluation rules into OCaml, we need to make a decision about how to handle this case. One straightforward approach is write the single-step evaluation function so that it raises an exception when none of the evaluation rules apply to the term that it is given. (Another possibility would be to make the single-step evaluator return a `term option` indicating whether it was successful and, if so, giving the resulting term; this would also work fine, but would require a little more bookkeeping.) We begin by defining the exception to be raised when no evaluation rule applies:

```
exception No
```

Now we can write the single-step evaluator itself. Note that there are several places where we are constructing terms from scratch rather than reorganizing existing terms. Since these new terms do not exist in the user's original source file, their `info` annotations are not useful. The constant `dummyinfo` is used as the `info` annotation in such terms.

```
let rec eval1 = function
    TmIf(fi,TmTrue(_),t2,t3) →
      t2
  | TmIf(fi,TmFalse(_),t2,t3) →
      t3
  | TmIf(fi,t1,t2,t3) →
      let t1' = eval1 t1 in
      TmIf(fi, t1', t2, t3)
  | TmSucc(fi,t1) →
      let t1' = eval1 t1 in
      TmSucc(fi, t1')
  | TmPred(_,TmSucc(_,v1)) when (isnumericval v1) →
      v1
  | TmPred(fi,t1) →
      let t1' = eval1 t1 in
      TmPred(fi, t1')
  | TmPred(_,TmZero(_)) →
      TmZero(dummyinfo)
  | TmIsZero(_,TmZero(_)) →
      TmTrue(dummyinfo)
  | TmIsZero(_,TmSucc(_,v11)) when (isnumericval v11) →
      TmFalse(dummyinfo)
  | TmIsZero(fi,t1) →
      let t1' = eval1 t1 in
```

```
        TmIsZero(fi, t1')
    | _ →
        raise No
```

Finally, the `eval` function takes a term and finds its normal form by repeatedly calling `eval1`. Whenever `eval1` returns a new term `t'`, we make a recursive call to `eval` to continue evaluating from `t'`. When `eval1` finally reaches a point where no rule applies, it raises the exception `No`, causing `eval` to break out of the loop and return the final term in the sequence.

```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with No → t
```

Obviously, this simple evaluator is tuned for easy comparison with the mathematical definition of evaluation, not for finding normal forms as quickly as possible. A somewhat more efficient algorithm can be obtained by starting instead from the big-step evaluation rules in Exercise **??**.

The Rest

Of course, there is much more to an interpreter or compiler—even a very simple one—than we are showing here.

```
           chars           tokens          ASTs              ASTs
file I/O   ----- >  lexer  ----- >  parser  --- >  evaluator  --- >  printer
```

We are concentrating here just on the evaluation step and the datatype of abstract syntax trees that are manipulated at this step.

Interested readers are encouraged to have a look at the on-line OCaml code for the whole typechecker.

3.4 Summary

Booleans $\mathbb{B}$ **(untyped)**

Syntax

`t` ::=		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v` ::=		*values*
	`true`	*value true*

`false` *value false*

Evaluation $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

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

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

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

Arithmetic expressions $\mathbb{B}$ $\mathbb{N}$ **(untyped)**

New syntactic forms

`t`	::=	...	*terms*
		`0`	*constant zero*
		`succ t`	*successor*
		`pred t`	*predecessor*
		`iszero t`	*zero test*
`v`	::=	...	*values*
		`nv`	*numeric value*
`nv`	::=	...	*numeric values*
		`0`	*zero value*
		`succ nv`	*successor value*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

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

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

$$\texttt{pred (succ nv)} \longrightarrow \texttt{nv} \qquad \text{(E-BETANATPS)}$$

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

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

$$\texttt{iszero (succ nv)} \longrightarrow \texttt{false} \qquad \text{(E-BETANATIS)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}_1'} \qquad \text{(E-IsZero)}$$

Chapter 4

The Untyped Lambda-Calculus

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

This chapter reviews the definition and some basic properties of the **untyped** or **pure lambda-calculus**, the underlying "computational substrate" for most of the type systems described in the rest of the book.

In the mid 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus capturing the language's 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. Most of the concepts and techniques we will develop for the lambda-calculus can be transferred quite directly to these 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 can add 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 lead eventually to languages such as ML [GMW79, MTH90, WAL+89, MTHM97], Haskell [HJW+92], or Scheme [SJ75, KCR98]. As we shall see in later chapters, extensions to the core language often involve extensions to the type system as well.

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 generically, in terms of one or more named parameters, and 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 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.

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 $\mathtt{t}_1$ to another lambda-term $\mathtt{t}_2$, written $\mathtt{t}_1\ \mathtt{t}_2$, is a lambda-term; and the abstraction of a variable `x` from a lambda-term $\mathtt{t}_1$, written $\lambda\mathtt{x}.\mathtt{t}_1$, is a lambda-term. These forms are summarized in the following grammar:

`t` ::=	*terms*
`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. In this book, we will use "expression" for all sorts of syntactic phrases (including type expressions, kind expressions, etc.), reserving "term" for the more specialized sense of phrases representing programs.

As we did for arithmetic expressions, we've defined here only the **abstract syntax** of lambda-terms, ignoring the processes of lexing and parsing through which strings of characters are mapped into abstract syntax trees. Our convention will be that application "associates to the left"—that is, `s t u` is a shorthand for `(s t) u`, i.e., for the tree

```
      (apply)
      |     |
   (apply)  u
    |   |
    s   t
```

and that the bodies of abstractions are assumed to extend as far to the right as possible, so that `λx. λy. x y x` means `λx. (λy. (x y) x)`, i.e.:

```
           λx
            |
           λy
            |
         (apply)
         |     |
     (apply)   x
      |   |
      x   y
```

The variable `x` is said to be **bound** in the body `t` of the abstraction `λx.t`. Conversely, we say that an occurrence of a variable `x` is **free** if `x` appears in a position 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))`. In the term `((λx.x) x)`, the variable `x` has both a free and a bound occurrence.

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;
```

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. Each step in the computation consists of rewriting an application whose left-hand component is an abstraction by substituting the right-hand part for the bound variable in the abstraction's body. Graphically,

$$(\lambda \mathtt{x}.\ \mathtt{t}_{12})\ \mathtt{t}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{t}_2\}\mathtt{t}_{12},$$

where $\{\mathtt{x} \mapsto \mathtt{t}_2\}\mathtt{t}_{12}$ means "the term obtained by replacing all free occurrences of `x` in $\mathtt{t}_{12}$ by $\mathtt{t}_2$." For example, `((λx.x) y)` rewrites to `y` and `((λx. x y) (u r))` rewrites to `(u r y)`. Following Church, a term of the form $((\lambda \mathtt{x}.\ \mathtt{t}_{12})\ \mathtt{t}_2)$ is called a **redex**, and the operation of rewriting a redex according to the above rule is called **beta-reduction**.

The lambda-calculus literature contains several variants of beta-reduction. The differences appear in the rules for the order in which redices may be reduced.

- Under **full beta-reduction**, *any* redex may be reduced at any time. This gives a nondeterministic notion of evaluation, where at each step we pick some redex, anywhere inside the term we are evaluating, and reduce it. For example, consider the term

  ```
  (λx.x) ((λx.x) (λz. (λx.x) z)),
  ```

or, more readably:

$$\texttt{id (id (}\lambda\texttt{z. id z)).}$$

This term contains three redices:

$$\begin{array}{l}
\underline{\texttt{id (id (}\lambda\texttt{z. id z))}}\\
\texttt{id (}\underline{\texttt{(id (}\lambda\texttt{z. id z))}}\texttt{)}\\
\texttt{id (id (}\lambda\texttt{z. }\underline{\texttt{id z}}\texttt{))}
\end{array}$$

Under full beta-reduction, we might choose, for example, to begin with the innermost redex, then reduce the one in the middle, and finish with the outermost:

$$\begin{array}{ll}
 & \texttt{id (id (}\lambda\texttt{z. }\underline{\texttt{id z}}\texttt{))}\\
\longrightarrow & \texttt{id }\underline{\texttt{(id (}\lambda\texttt{z.z))}}\\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z.z)}}\\
\longrightarrow & \lambda\texttt{z.z}\\
\not\longrightarrow &
\end{array}$$

- Under the **normal order** reduction strategy, the leftmost, outermost redex is always reduced first. Under this strategy, the term above would be reduced as follows:

$$\begin{array}{ll}
 & \underline{\texttt{id (id (}\lambda\texttt{z. id z))}}\\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z. id z)}}\\
\longrightarrow & \lambda\texttt{z. }\underline{\texttt{id z}}\\
\longrightarrow & \lambda\texttt{z.z}\\
\not\longrightarrow &
\end{array}$$

- The **call by name** (or **lazy**) reduction strategy is yet more restrictive, allowing no reductions under abstractions. Starting from the same term, we would perform the first two reductions as under normal-order, but then stop before the last and regard λ`z. id z` as a normal form:

$$\begin{array}{ll}
 & \underline{\texttt{id (id (}\lambda\texttt{z. id z))}}\\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z. id z)}}\\
\longrightarrow & \lambda\texttt{z. id z}\\
\not\longrightarrow &
\end{array}$$

Variants of call by name have been used in a few popular programming languages, notably Algol [**?**] and Haskell [HJW+92] (Haskell actually uses an optimized version known as **call by need**).

- Most programming languages today use a **call by value** reduction strategy, in which a redex is allowed to reduce only when its right-hand side has already been reduced to a **value**, where a value in this setting is just an abstraction. Under this strategy, our example term reduces as follows:

$$
\begin{array}{ll}
 & \texttt{id}\ \underline{\texttt{(id (}\lambda\texttt{z. id z))}} \\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z. id z)}} \\
\longrightarrow & \lambda\texttt{z. id z} \\
\not\longrightarrow &
\end{array}
$$

Some authors use the terms "reduction" and "evaluation" synonymously. Others use "evaluation" only for strategies that involve some notion of "value" (call by name and call by value), and "reduction" for nondeterministic relations.

For most of the development of type systems and their properties, the choice of reduction strategy makes little difference. The typing issues that arise, and the techniques used to address them, are much the same for all the strategies. For the rest of this book, we'll use call by value, which will make it easier to discuss extensions such as references (Chapter 11) and exceptions (Chapter 10).

4.2 Programming in the Lambda-Calculus

The lambda-calculus is much more powerful than its tiny definition might suggest. To illustrate this point, we develop a number of standard examples of programming in the lambda-calculus. These examples are not intended to suggest that the lambda-calculus should be taken as a full-blown programming language in its own right—most full-blown languages provide clearer and more efficient ways of accomplishing the same tasks—but rather as warm-up exercises to get the feel of the system.

As a first example, note that there is no built-in provision in the lambda-calculus for multi-argument functions. Of course, this would not be hard to add, 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 that 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 $\texttt{((}\lambda\texttt{y.}\{\texttt{x} \mapsto \texttt{p}\}\texttt{s) q)}$ and then to $\{\texttt{y} \mapsto \texttt{q}\}\{\texttt{x} \mapsto \texttt{p}\}\texttt{s}$. This transformation of multi-argument functions into higher-order functions is often called **Currying** after its popularizer, Haskell Curry. (It was actually invented by Frege.)

Another 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 `n` 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:

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

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:

	`fst (pair m n)`	
$=$	`fst ((λf. λs. λb. b f s) m n)`	by definition
$\longrightarrow$	`fst ((λs. λb. b m s) n)`	reducing the underlined redex
$\longrightarrow$	`fst (λb. b m n)`	reducing the underlined redex
$=$	`(λp. p tru) (λb. b m n)`	by definition
$\longrightarrow$	`(λb. b m n) tru`	reducing the underlined redex
$\longrightarrow$	`tru m n`	reducing the underlined redex
$\longrightarrow^*$	`m`	as before.

The encoding of numbers as lambda-terms is only slightly more intricate than what we have just seen. Define the **Church numerals** c_0, c_1, c_2, etc., as follows:

```
c₀ = λs. λz. z;
c₁ = λs. λz. s z;
c₂ = λs. λz. s (s z);
c₃ = λs. λz. s (s (s z));
c₄ = λs. λz. s (s (s (s z)));
etc.
```

That is, each number n is represented by a combinator c_n that takes two arguments, `s` and `z` (for "successor" and "zero"), 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:

```
plus = λm. λn. λs. λz. m s (n s z);
times = λm. λn. m (plus n) c₀;
```

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.

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 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."

4.2.2 Exercise [Recommended]: Define a similar term for calculating the successor of a number. Solution on page 282. □

4.2.3 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.

4.2.4 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)? □

4.2.5 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 283. □

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques.

4.2.6 Exercise [Homework]: *(Due on Feb. 8.)* 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. (There are other ways of encoding lists besides this one. You may enjoy trying to find another.) Solution on page 283. □

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 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`.[1]

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

Suppose we want to write a recursive function definition of the form `ff` $=$ ⟨*body containing* `ff`⟩—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 42. 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) * ...))
```

[1]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

```
fixedpoint_n = λf. (λx. f (x x)) (λx. f (x x));
```

is useless in a call-by-value setting, since an expression like `fixedpoint_n ff` always diverges, no matter what `ff` is.

This effect can be achieved using `fixedpoint` by defining gg = `λf.`⟨*body containing* `f`⟩ and `ff` = `fixedpoint gg`. For example, we can define the factorial function by

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

We then have:

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

4.2.7 Exercise [Quick check]: Why did we use `equal` here to verify the result returned by `factorial` instead of just computing the normal form of `factorial` c_3 and printing it out? Solution on page 283. □

4.2.8 Exercise [Recommended]: Use the `fixedpoint` combinator and the encoding of lists from Exercise 4.2.6 to write a function that sums lists of church numerals. Solution on page 284. □

4.3 Formalities

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

Syntax

As in Chapter 3, the abstract grammar defining terms should be read as shorthand for an inductively defined set of abstract syntax trees.

4.3.1 Definition [Terms]: Let $\mathcal{V}$ be a countable set of variable names. 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 $\mathtt{x} \in \mathcal{V}$, 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 `t` can be defined exactly as we did for arithmetic expressions in Definition 3.2.9. More interestingly, we can give a simple inductive definition of the set of variables appearing free in a lambda-term.

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

$$\begin{array}{lcl} FV(\mathtt{x}) & = & \{\mathtt{x}\} \\ FV(\lambda\mathtt{x.t_1}) & = & FV(\mathtt{t_1}) \setminus \{\mathtt{x}\} \\ FV(\mathtt{t_1\ t_2}) & = & FV(\mathtt{t_1}) \cup FV(\mathtt{t_2}) \end{array}$$ □

4.3.3 Exercise: Give a detailed proof that $|FV(\mathtt{t})| \leq size(\mathtt{t})$ for every term $\mathtt{t}$. Solution on page 284. □

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 Chapter 5 we'll go the rest of the way, defining a more refined "nameless" presentation of terms on which substitution is completely formal. The reason for presenting both is that the nameless formulation is easier to implement correctly but too fiddly to use in definitions and proofs. Having worked it out in detail, we can 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:

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

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

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

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 names of bound variables do not matter*—the identity function is exactly the same whether we write it $\lambda\mathtt{x.x}$ or $\lambda\mathtt{y.y}$ or $\lambda\mathtt{franz.franz}$. 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 $\mathtt{x}$ in a term $\mathtt{t}$ (which should get replaced during substitution) and *bound* ones, which should not. When we reach an abstraction binding the name $\mathtt{x}$ inside of $\mathtt{t}$, the substitution operation should stop. This leads to the next attempt at a definition of substitution:

$$
\begin{array}{lcll}
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{x} & = & \mathtt{s} & \\
\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y.t_1}) & = & \left\{ \begin{array}{l} \lambda\mathtt{y.\ t_1} \\ \lambda\mathtt{y.}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} \end{array} \right. & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \end{array} \\
\{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t_1\ t_2}) & = & \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_2} &
\end{array}
$$

This is better, but it is still not quite right. For example, consider what happens when we substitute the term $\mathtt{z}$ for the variable $\mathtt{x}$ in the term $\lambda\mathtt{z.x}$:

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

This time, we have made essentially the opposite mistake as in the previous unfortunate example: we've turned the constant function $\lambda\mathtt{z.x}$ into the identity function! Again, this occurred only because we happened to choose $\mathtt{z}$ 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 $\mathtt{s}$ becoming bound when $\mathtt{s}$ is naively substituted into a term $\mathtt{t}$ containing variable binders is called **variable capture**. To avoid it, we need to make sure that the bound variable names of $\mathtt{t}$ are kept distinct from the free variable names of $\mathtt{s}$. 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:

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

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," in Church's terminology):

4.3.4 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 arguments for which it is undefined, we can rename 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 `y` is different from both `x` and the free variables of `s`. This yields the final form of the definition.

4.3.5 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 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 5.1.

4.3.6 Exercise: 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? □

Define multi-substitutions. Give enough examples to answer questions about idempotency, left-to-rightness, etc. Mention Barendregt convention.

Operational Semantics

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

`v` ::=		*values*
	$\lambda\mathtt{x.t}$	*abstraction value*

□

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

$$(\lambda x.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

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

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

As in the definition of the evaluation relation for arithmetic expressions, there are two sorts of rules here: the *computation* rule E-BETA and the *congruence* rules E-APP1 and E-APP2. □

Summary

λ : Untyped lambda-calculus $\rightarrow$ **(untyped)**

Syntax

t ::=		*terms*
	x	*variable*
	$\lambda x.t$	*abstraction*
	$t\ t$	*application*
v ::=		*values*
	$\lambda x.t$	*abstraction value*

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

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

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

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]).

Chapter 5

Implementing the Lambda-Calculus

Just because you've implemented something doesn't mean you understand it.
— Brian Cantwell Smith

5.1 Nameless Representation of Terms

In the previous chapter, we worked with terms "up to renaming of bound variables." This is fine for discussing basic concepts and for presenting proofs cleanly, but for building an implementation we need to choose a single concrete representation for each term.

There are various ways to do this:

1. We can represent a variable occurrence as a string. This makes our concrete ("algorithmic") representation close to the declarative one, but it means that we have to alpha-convert during substitution. This becomes quite tricky and it is easy to introduce subtle bugs.

2. We can replace the names chosen by the programmer with some canonical naming scheme.

3. We can avoid substitution altogether by introducing mechanisms like closures.

We choose the second, using a well-known technique due to Nicolas de Bruijn [dB72]. Making this choice requires some work now, but once the basic ideas are understood, their realization in the implementations throughout the book will be completely mechanical.

Syntax

De Bruijn's idea was that we can represent terms more straightforwardly—if less readibly—by making variable occurrences *point directly to* their binders, rather than referring to them by name. This can be accomplished by replacing named variables by natural numbers, where the number `k` stands for "the variable bound by the `k`'th enclosing λ." For example, the ordinary term `λx.x` corresponds to the **nameless term** `λ.0`, while `(λx.λy. x (y x))` is represented by the nameless term `(λ.λ. 1 (0 1))`. Nameless terms are also sometimes called **de Bruijn terms**, and the numeric variables in them are called **de Bruijn indices**.[1]

5.1.1 Exercise [Quick check]: For each of the following combinators

```
c0 = λz. λs. z;
c2 = λz. λs. s (s z);
plus = λm. λn. λz. λs. m (n z s) s;
fixedpoint = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));
foo = (λx. (λx. x)) (λx. x);
```

write down the corresponding nameless term. □

Note that each (closed) ordinary term has just one de Bruijn representation, and that two ordinary terms are equivalent modulo renaming of bound variables iff they have the same de Bruijn representation.

To deal with terms containing free variables, we need the idea of a naming context. For example, suppose we want to represent `(λx. y x)` as a nameless term. We know what to do with `x`, but we cannot see the binder for `y`, so it is not clear how "far away" it might be and we do not know what number to assign to it. The solution is to choose, once and for all, an assignment (called a **naming context**) of de Bruijn indices to free variables, and use this assignment consistently when we need to choose numbers for free variables. For example, suppose that we choose to work under the following naming context:

$$\begin{array}{rcl} \Gamma & = & \mathtt{x} \mapsto 4 \\ & & \mathtt{y} \mapsto 3 \\ & & \mathtt{z} \mapsto 2 \\ & & \mathtt{a} \mapsto 1 \\ & & \mathtt{b} \mapsto 0 \end{array}$$

Then `(x (y z))` would be represented as `(4 (3 2))`, while `(λx. y x)` would be represented as `(λ. 4 0)` and `(λw.λa.x)` would be represented as `(λ.λ.6)`.

Actually, all we need to know about Γ is the order in which the variables appear. Instead of writing it in tabular form, as we did above, we will elide the numbers and write it as a sequence.

[1] Note on pronunciation: the nearest English approximation to the second syllable in "de Bruijn" is "brown," not "broyn."

5.1.2 Definition: Suppose $\mathtt{x}_0$ through $\mathtt{x}_n$ are variable names from $\mathcal{V}$. The naming context $\Gamma = \mathtt{x}_n, \mathtt{x}_{n-1}, \ldots \mathtt{x}_1, \mathtt{x}_0$ assigns to each $\mathtt{x}_i$ the de Bruijn index i. Note that the rightmost variable in the sequence is given the index 0; This matches the way we count λs—from right to left—when converting a named term to nameless form. We write $dom(\Gamma)$ for the set $\{\mathtt{x}_0, \ldots, \mathtt{x}_n\}$ of variable names mentioned in Γ. □

Formally, we define the syntax of nameless terms almost exactly like the syntax of ordinary terms (cf. Definition 4.3.1). The only difference is that we need to keep careful track of how many free variables each term may contain. That is, we distinguish the sets of terms with no free variables (called the 0-terms), terms with at most one free variable (1-terms), and so on.

5.1.3 Definition [Terms]: Let $\mathcal{T}$ be the smallest family of sets $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$ such that

1. $\mathtt{k} \in \mathcal{T}_n$ whenever $0 \leq \mathtt{k} < n$;
2. if $\mathtt{t} \in \mathcal{T}_n$ and $n > 0$, then $\lambda.\mathtt{t} \in \mathcal{T}_{n-1}$;
3. if $\mathtt{t}_1 \in \mathcal{T}_n$ and $\mathtt{t}_2 \in \mathcal{T}_n$, then $(\mathtt{t}_1\ \mathtt{t}_2) \in \mathcal{T}_n$.

The elements of each $\mathcal{T}_n$ are called n-terms. □

Note that the elements of $\mathcal{T}_n$ are terms with *at most* n free variables numbered between 0 and $n-1$: a given element of $\mathcal{T}_n$ need not have free variables with all these numbers, or indeed any free variables at all. If $\mathtt{t}$ is closed, for example, it will be an element of $\mathcal{T}_n$ for every n.

Also note that, strictly speaking, it does not make sense to speak of "some $\mathtt{t} \in \mathcal{T}$"—we always need to specify how many free variables $\mathtt{t}$ might have. In practice, though, we will often have some fixed naming context Γ in mind; we will then abuse the notation slightly and write $\mathtt{t} \in \mathcal{T}$ to mean $\mathtt{t} \in \mathcal{T}_n$, where n is the length of Γ.

5.1.4 Exercise: Give an alternative construction of the sets of n-terms in the style of 3.2.3, and show (as we did in Proposition 3.2.6) that it is equivalent to the one above. □

5.1.5 Exercise [Recommended]:

1. Define a function *removenames* that takes a naming context Γ and an ordinary term $\mathtt{t}$ (with $FV(\mathtt{t}) \subseteq dom(\Gamma)$) and yields the corresponding nameless term.
2. Define a function *restorenames* that takes a nameless term $\mathtt{t}$ and a naming context Γ and produces an ordinary term. (To do this, you will need to "make up" names for the variables bound by abstractions in $\mathtt{t}$. You may assume that the set $\mathcal{V}$ of variable names is ordered, so that it makes sense to say "choose the first variable name that is not already in $dom(\Gamma)$.")

This pair of functions should have the property that

$$\textit{removenames}_\Gamma(\textit{restorenames}_\Gamma(\mathtt{t})) = \mathtt{t}$$

for any nameless term t, and similarly

$$\textit{restorenames}_\Gamma(\textit{removenames}_\Gamma(\mathtt{t})) = \mathtt{t},$$

up to renaming of bound variables, for any ordinary term t. □

Shifting and Substitution

Before defining substitution on nameless terms, we need one auxiliary operation on terms, called "shifting," which renumbers the indices of the free variables in a term. When a substitution goes under a λ-abstraction, as in $\{\mathtt{x} \mapsto \mathtt{s}\}(\lambda\mathtt{y.x})$, the context in which the substitution is taking place becomes one variable longer than the original; we need to increment the indices of the free variables in s so that they keep referring to the same names in the new context as they did before.

5.1.6 Definition [Shifting]: The d-place shift of a term t above cutoff c, written $\uparrow^d_c(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl}
\uparrow^d_c(\mathtt{k}) & = & \begin{cases} \mathtt{k} & \text{if } \mathtt{k} < c \\ \mathtt{k} + d & \text{if } \mathtt{k} \geq c \end{cases} \\
\uparrow^d_c(\lambda.\mathtt{t}_1) & = & \lambda.\ \uparrow^d_{c+1}(\mathtt{t}_1) \\
\uparrow^d_c(\mathtt{t}_1\ \mathtt{t}_2) & = & \uparrow^d_c(\mathtt{t}_1)\ \ \uparrow^d_c(\mathtt{t}_2)
\end{array}$$

We write $\uparrow^d(\mathtt{t})$ for $\uparrow^d_0(\mathtt{t})$. □

5.1.7 Exercise [Quick check]:

1. What is $\uparrow^2(\lambda.\lambda.\ \mathtt{1}\ (\mathtt{0}\ \mathtt{2}))$?
2. What is $\uparrow^2(\lambda.\ \mathtt{0}\ \mathtt{1}\ (\lambda.\ \mathtt{0}\ \mathtt{1}\ \mathtt{2}))$? □

5.1.8 Exercise: Show that if t is an n-term, then $\uparrow^d_c(\mathtt{t})$ is an $(n+d)$-term. □

Now we are ready to define the substitution operator $\{j \mapsto \mathtt{s}\}\mathtt{t}$. Later on we will be most interested in substituting for the *last* variable in the context (i.e., $j = 0$), since that is the case we need in order to define the operation of beta-reduction. However, to substitute for variable 0 in a term that happens to be a λ-abstraction, we need to be able to substitute for the variable number numbered 1 in its body. Thus, the definition of substitution must work on an arbitrary variable.

5.1.9 Definition [Substitution]: The substitution of a term $\mathtt{s}$ for variable number $\mathtt{j}$ in a term $\mathtt{t}$, written $\{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{t}$, is defined as follows:

$$
\begin{array}{lcl}
\{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{k} & = & \begin{cases} \mathtt{s} & \text{if } \mathtt{k} = \mathtt{j} \\ \mathtt{k} & \text{otherwise} \end{cases} \\
\{\mathtt{j} \mapsto \mathtt{s}\}(\lambda.\mathtt{t}_1) & = & \lambda.\ \{\mathtt{j}+1 \mapsto \uparrow^1(\mathtt{s})\}\mathtt{t}_1 \\
\{\mathtt{j} \mapsto \mathtt{s}\}(\mathtt{t}_1\ \mathtt{t}_2) & = & (\{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{t}_1\ \{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{t}_2)
\end{array}
$$
□

5.1.10 Exercise [Quick check]: Convert the following substitution operations to nameless form, assuming the global context is $\Gamma = \mathtt{a},\mathtt{b}$, and calculate their results using the above definition. Do the answers correspond to the original definition of substitution on ordinary terms in Section 4.3?

- $\{\mathtt{b} \mapsto \mathtt{a}\}$ `(b (λx.λy.b))`
- $\{\mathtt{b} \mapsto$ `(a (λz.a))`$\}$ `(b (λx.b))`
- $\{\mathtt{b} \mapsto \mathtt{a}\}$ `(λb. b a)`
- $\{\mathtt{b} \mapsto \mathtt{a}\}$ `(λa. b a)` □

5.1.11 Exercise: Show that if $\mathtt{s}$ and $\mathtt{t}$ are n-terms and $\mathtt{j} \leq n$, then $\{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{t}$ is an n-term. □

5.1.12 Exercise [Quick check]: Take a sheet of paper and, without looking at the definitions of substitution and shifting above, see if you can regenerate them. □

5.1.13 Exercise [Homework]: *(Due on Feb. 10.)* The definition of substitution on nameless terms should agree with our informal definition of substitution on ordinary terms. $\Leftarrow$

1. What theorem needs to be proved to justify this correspondence rigorously?
2. Prove it.

Solution on page 284. □

Evaluation

We can now define the evaluation relation on nameless terms.

The beta-reduction rule is defined in terms of our new nameless substitution operation. The only slightly subtle technical point is that reducing a redex "uses up" the bound variable—when we reduce $((\lambda\mathtt{x}.\mathtt{t}_1)\ \mathtt{v}_2)$ to $\{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_1$, the bound variable $\mathtt{x}$ disappears in the process. Thus, we will need to renumber the variables of the result of substitution to take into account the fact that $\mathtt{x}$ is no longer part of the context. Similarly, we need to shift the variables in $\mathtt{v}_2$ up by one before substituting into $\mathtt{t}_1$ to take account of the fact that $\mathtt{t}_1$ is defined in a larger context than $\mathtt{v}_2$.

$$(\lambda.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \uparrow^{-1}(\{0 \mapsto \uparrow^1(\mathtt{v}_2)\}\mathtt{t}_{12}) \qquad \text{(E-BETA)}$$

The other rules are identical to what we had before.

5.2 A Concrete Realization

An executable evaluator for untyped lambda-terms can be obtained by a straightforward translation of the foregoing definitions into OCaml. As in Section 3.3, we show just the core algorithms, ignoring issues of lexing, parsing, the top-level read-eval-print loop, etc.

Syntax

We can obtain a datatype representing abstract syntax trees for terms by directly transliterating Definition 5.1.3:

```
type term =
    TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

The representation of a variable is a number—its de Bruijn index. The representation of an abstraction carries just a subterm for the abstraction's body. An application carries the two subterms being applied.

The definition actually used in our implementation, however, will carry a little bit more information. First, as before, it is useful to annotate every term with an element of the type `info` recording the file position where that term was originally found, so that error printing routines can direct the user (or even the user's text editor, automatically) to the precise point where the error occurred.

```
type term =
    TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Second, for purposes of debugging, it is helpful to carry an extra number on each variable node, as a consistency check. The convention will be that this second number will always contain the *total length* of the context in which the variable is defined.

```
type term =
    TmVar of info * int * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Whenever a variable is printed, we will verify that this number corresponds to the actual size of the current context; if it does not, then a shift operation has been forgotten someplace.

The final refinement also concerns printing. Although terms are represented internally using de Bruijn indices, this is certainly not how they should be presented to the user: we should convert from the ordinary representation to nameless terms during parsing, and convert back to ordinary form during printing. There is nothing hard about doing this, but if we do it naively, users will not be very happy, since the names of the bound variables in the terms that are printed will have nothing to do with the names in the original program. This can be fixed by annotating each abstraction with a string to be used as a hint for the name of the bound variable.

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

The basic operations on terms (substitution in particular) do not do anything fancy with these strings: they are simply carried along in their original form, with no checks for name clashes, capture, etc. When the printing routine needs to generate a fresh name for a bound variable, it tries first to use the supplied hint; if this turns out to clash with a name already used in the current context, it tries similar names, adding primes until it finds one that is not currently being used. This ensures that the printed term will be similar to what the user expects, modulo a few primes.

The printing routine itself looks like this:

```
let rec prtm ctx = function
    TmAbs(fi,x,t1) →
      let (x',ctx') = pickfreshname ctx x in
      pr "(lambda "; pr x'; pr ". "; prtm ctx' t1; pr ")"
  | TmApp(fi, t1, t2) →
      pr "("; prtm ctx t1; pr " "; prtm ctx t2; pr ")"
  | TmVar(fi,x,n) →
      if ctxlength ctx = n then
        pr (index2name fi ctx x)
      else
        pr "[bad index]"
```

It uses several lower-level functions: `pr` sends a string to the terminal; `ctxlength` returns the length of a context; and `index2name` looks up the string name of a variable from its index. The most interesting one is `pickfreshname`, which takes a context `ctx` and a string hint `x`, finds a name `x'` similar to `x` such that `x'` is not already listed in `ctx`, adds `x'` to `ctx` to form a new context `ctx'`, and returns both `x'` and `ctx'`.

(The actual printing function found in the on-line implementation is somewhat more complicated than this one, taking into account two additional issues. First, it leaves out as many parentheses as possible, following the conventions that application associates to the left and the bodies of abstractions extend as far to the right as possible. Second, it generates formatting instructions for a low-level *prettyprinting* layer that makes decisions about line breaking and indentation.)

Shifting and Substitution

The definitions of shifting (5.1.6) can be translated almost symbol for symbol into OCaml:

```
let tmshift d t =
  let rec walk c = function
    TmVar(fi,x,n) → if x>=c then TmVar(fi,x+d,n+d) else TmVar(fi,x,n+d)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

The internal shifting $\uparrow^{\mathtt{d}}_{\mathtt{c}}(\mathtt{t})$ is here represented by a call to the inner function `(walk c t)`. Since `d` never changes, there is no need to pass it along to each call to `walk`; we just use the outer binding of `d` when we need it in the variable case of `walk`. The top-level shift $\uparrow^{\mathtt{d}}(\mathtt{t})$ is represented by `(tmshift d t)`.

Similarly, the function `subst` comes almost directly from Definition 5.1.9:

```
let tmsubst j s t =
  let rec walk c = function
    TmVar(fi,x,n) → if x=j then (tmshift j s) else TmVar(fi,x,n)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

The substitution of term `s` for the variable numbered `j` in term `t`, formerly written $\{\mathtt{j} \mapsto \mathtt{s}\}\mathtt{t}$, is represented as `(tmsubst j s t)` here. (The only difference from the original definition of substitution is that here we do all the shifting of `s` at once, in the `TmVar` case, rather than shifting `s` up by one every time we go through a binder. This means that the argument `j` is the same in every call to `walk`, and we can omit it from the inner definition.)

(The reader may note that the definitions of `tmshift` and `tmsubst` are very similar, differing only in the action that is taken when a variable is reached. The actual on-line imlementation exploits this observation to express both shifting and substitution operations as special cases of a more general function called `tmmap`. Given a term `t` and a function `onvar`, the result of `(tmmap onvar t)` is a term of the same shape as `t` but where every variable is replaced by the result of calling `onvar` on that variable. This saves quite a bit of tedious repetition in some of the larger calculi.)

In the operational semantics of the lambda-calculus, the only place where substitution is used is in the beta-reduction rule. As we noted before, this rule actually performs several operations: the term being substituted for the bound variable is first shifted up by one, then the substitution is made, and then the whole result is shifted down by one to account for the fact that the bound variable has been used up. The following definition encapsulates this sequence of steps:

```
let tmsubstsnip s t = tmshift (-1) (tmsubst 0 (tmshift 1 s) t)
```

Evaluation

As in Chapter 3, the evaluation function depends on an auxiliary predicate `isval`:

```
let rec isval ctx = function
    TmAbs(_,_,_) → true
  | _ → false
```

The single-step evaluation function is a direct transcription of the evaluation rules, except that we pass a context `ctx` along with the term. This argument is never used in the present evaluator, but it is needed by some of the more complex evaluators later on.

```
let rec eval1 ctx = function
    TmApp(fi,TmAbs(_,x,t11),v2) when (isval ctx v2) →
      tmsubstsnip v2 t11
  | TmApp(fi,v1,t2) when (isval ctx v1) →
      let t2' = eval1 ctx t2 in
      TmApp(fi, v1, t2')
  | TmApp(fi,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmApp(fi, t1', t2)
  | _ →
      raise No
```

The multi-step evaluation function is the same as before, except for the `ctx` argument:

```
let rec eval ctx t =
  try let t' = eval1 ctx t
      in eval ctx t'
  with No → t
```

5.3 Ordinary vs. Nameless Representations

Strictly speaking, having gone to the trouble of making all this so precise, we should now continue through the rest of the book using de Bruijn representations consistently throughout. However, we will not, as it would make even simple definitions nearly impenetrable. Instead, we will adopt the following compromise:

- In OCaml code, we use de Bruijn representations.
- In mathematical presentations of calculi, discussions, examples, and proofs, we use the original presentation in terms of named variables. However, we always regard this as just a convenient shorthand for an underlying de Bruijn representation.

  This convention has the following consequences:

1. terms are always understood modulo renaming of bound variables, and
2. whenever we mention a term, we will always (at least implicitly) have some context in mind that imposes a numerical ordering on its free variable names.

Chapter 6

Typed Arithmetic Expressions

Recall the syntax for terms in our simple calculus of arithmetic expressions:

`t ::=`		*terms*
	`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*

We saw in Chapter 3 that evaluating a term can either result in a *value*

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

or else get *stuck* at some stage, by reaching a term like `(pred false)`.

We would like to be able to tell, without actually evaluating a term, that its evaluation will definitely *not* get stuck. To do this, we need to be able to distinguish between terms whose result will be a numeric value (since these are the only ones that should appear as arguments to `pred`, `succ`, and `iszero`) and terms whose result will be a boolean (since only these should appear as the guard of a conditional).

We introduce two classifiers: `Bool` and `Nat`. Saying that "a term `t` has type `T`" (or "belongs to `T`," or "is an element of `T`") means that `t` "obviously" evaluates to a value of the appropriate form—where by "obviously" we mean that we can see this *statically*, without doing any evaluation of `t`. For example, the term `(if true then false else true)` has type `Bool`, while the term `(pred (succ (pred (succ 0))))` has type `Nat`. However, our analysis of the types of terms will be **conservative**, only making use of static information. This means that we will not be able to conclude that terms like `(if true then 0 else false)` have any type at all even though their evaluation does not get stuck.

6.1 Syntax

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

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

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

6.2 The Typing Relation

The type system for arithmetic expressions is defined by a set of rules assigning types to terms. We write "`t : 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:

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

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

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

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

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

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

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

The boolean constants both have type `Bool`

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

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

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

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

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).

6.3 Properties of Typing and Reduction

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 `t : T`, pronounced "the term `t` has the type `T`." (In the literature on type systems, the symbol $\in$ is often used instead of `:` for the typing relation.) The provable statements of this logic are defined as follows:

6.3.1 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. □

6.3.2 Definition: A **typing derivation** is a tree of instances of the typing rules. Each statement `t : T` in the typing relation is justified by a typing derivation with conclusion `t : 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**. □

6.3.3 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 $(\mathsf{P}(\mathcal{J}')$, for all $\mathcal{D}' :: \mathcal{J}'$ with $\mathit{size}(\mathcal{D}') < \mathit{size}(\mathcal{D}))$ implies $\mathsf{P}(\mathcal{J})$, then we may conclude that $\mathsf{P}(\mathcal{J})$ holds for every derivable typing statement. □

Typechecking

6.3.4 Lemma [Inversion of the typing relation]:

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

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

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

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

6.3.5 Theorem [Uniqueness of Types]: Each term $\mathtt{t}$ has at most one type. That is, if $\mathtt{t}$ is typeable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules of Definition 6.3.1. □

Proof: Exercise. □

Safety = Preservation + Progress

The most basic property of this type system or any other is **safety** (or **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.

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

Proof: Exercise. □

The preservation theorem is often called "subject reduction" (or "subject evaluation") in the literature—the intuition being that a typing statement `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 that a "stuck" term is one that is in normal form but is not a value (Definition 3.2.22).

6.3.7 Theorem [Progress]: If `t : T`, then `t` is not stuck—i.e., either `t` is a value or there is some `t'` such that $\texttt{t} \longrightarrow \texttt{t}'$. □

Proof: Exercise. □

The combination of the latter property with the preservation 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."

6.3.8 Exercise [Homework]: *(Due on Feb. 15.)* Having seen the "subject reduction" property, we may wonder whether the opposite ("subject *expansion*") property also holds. Is it always the case that, if $\texttt{t} \longrightarrow^* \texttt{t}'$ and `t' : T`, then `t : T`? If so, prove it. If not, give a counterexample. Solution on page 284. □ ⇐

6.4 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"
```

6.5 Summary

Typing rules for booleans $\mathbb{B}$ **(typed)**

New syntactic forms

`T`	::=	...	*types*
		`Bool`	*type of booleans*

New typing rules $\boxed{\texttt{t : T}}$

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

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

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

Typing rules for numbers $\mathbb{B}$ $\mathbb{N}$ **(typed)**

New syntactic forms

`T`	::=	...	*types*
		`Nat`	*type of natural numbers*

New typing rules $\boxed{\texttt{t : T}}$

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

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

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

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

Chapter 7

Simply Typed Lambda-Calculus

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

7.1 Syntax

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

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

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

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

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

□

In the literature on type systems, two different presentation styles 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.

7.2 The Typing Relation

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

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

Since, in general, function abstractions can be nested, typing assertions actually have the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, pronounced "term $\mathtt{t}$ has type $\mathtt{T}$ 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 $\mathtt{x}$ be chosen so that it does not already appear in $dom(\Gamma)$; 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 write $dom(\Gamma)$ for the set of variables bound by Γ.

So the rule for typing abstractions actually has the general form

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

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:

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

Next, we need a rule for application:

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

In English: If $\mathtt{t}_1$ evaluates to a function mapping arguments in $\mathtt{T}_{11}$ to results in $\mathtt{T}_{12}$ (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 $\mathtt{T}_{11}$, then the result of applying $\mathtt{t}_1$ to $\mathtt{t}_2$ will be a value of type $\mathtt{T}_{12}$.

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:

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

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

1. `f:Bool→Bool` $\vdash$ `f (if false then true else false) : Bool`

2. `f:Bool→Bool` $\vdash$ `λx:Bool. f (if x then false else x) : Bool→Bool` □

7.2.2 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** (T-ABS) describing how elements of the type can be *created*, and
2. an **elimination rule** (T-APP) 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 for each type constructor we consider.

7.2.3 Exercise [Quick check]: Which of the rules for the type `Bool` are introduction rules and which are elimination rules? Solution on page 285. □

7.3 Summary

We have been discussing the simply typed lambda-calculus with booleans and conditionals. 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 with no base types at all, and a separate set of rules for booleans (which we have already seen on page 72).

$\lambda^{\rightarrow}$: Simply typed lambda-calculus $\rightarrow$ **(typed)**

Syntax

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

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$(\lambda\mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \{\mathtt{x}\mapsto\mathtt{v}_2\}\mathtt{t}_{12} \qquad \text{(E-BETA)}$$

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

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-APP2)}$$

Typing $\boxed{\Gamma\vdash\mathtt{t} : \mathtt{T}}$

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

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

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

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.

7.3.1 Exercise [Quick check]: The system $\lambda^{\to}$ by itself is actually trivial, in the sense that it has no well-typed programs at all. Why? Solution on page 285. □

7.4 Properties of Typing and Reduction

Typechecking

As in Chapter 6, 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 7.4.8).

7.4.1 Lemma [Inversion of the typing relation]:

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

7.4.2 Exercise [Easy]: Write out a typechecking algorithm for $\lambda^{\to}$ in pseudo-code or a programming language of your choice. Compare your answer with the OCaml implementation in Section 7.5. □

In Section 7.1, 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.)

7.4.3 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 that generate the typing relation. □

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 $\mathtt{x}:\mathtt{T} \in \Gamma$

By case (1) of the inversion lemma (7.4.1), 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. □

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

7.4.5 Exercise [Homework]: *(Due on Feb. 15.)* Is there any context Γ and type $\mathtt{T}$ such that $\Gamma \vdash \mathtt{x\ x} : \mathtt{T}$? If so, give Γ and $\mathtt{T}$ and show a typing derivation for $\Gamma \vdash \mathtt{x\ x} : \mathtt{T}$; if not, prove it. □ $\Leftarrow$

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.

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

Proof: Straightforward induction on typing derivations. □

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

Proof: Straightforward induction on typing derivations. □

Typing and Substitution

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

7.4.9 Exercise [Recommended]: Prove the substitution lemma, using an induction on the depth of typing derivations and Lemma 7.4.1. Solution on page 285. □

Type Safety

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

Proof: Exercise. □

7.4.11 Exercise [Homework]: *(Due on Feb. 15.)* In Exercise 6.3.8 we investigated the **subject expansion** property for our simple calculus of typed arithmetic expressions. Does it hold for the "functional part" of the simply typed lambda-calculus? That is, suppose `t` does not contain any conditional expressions. Do $\Gamma \vdash \texttt{t} \longrightarrow^* \texttt{t}'$ and $\texttt{t}' : \texttt{T}$ imply $\Gamma \vdash \texttt{t} : \texttt{T}$? Solution on page 286. □ $\Longleftarrow$

7.4.12 Theorem [Progress]: Suppose `t` is a closed normal form. If $\vdash \texttt{t} : \texttt{T}$, then `t` is a value. □

Proof: Outline: first show that every closed value of type `Bool` is either `true` or `false` and every closed value of type $\texttt{S}\to\texttt{T}$ is a λ abstraction... □

Erasure and Typeability

Untyped lambda-calculus; the erasure function; erasure as "compilation" to an untyped underlying machine. Definition of typeability of untyped terms; simple examples.

7.5 Implementation

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

type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term

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

let rec typeof ctx t =
  match t with
    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, tyT)
  | 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")
  | 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"
```

7.6 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 [Hin97]. 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 8

Normalization

In this chapter, we consider another interesting property of the simply typed lambda-calculus: the fact that all well-typed programs halt in a finite number of steps. This property is arguably of small pragmatic importance, since all real programming languages include constructs like general recursion (Section 9.9) or recursive types (Chapter 15) that can be used to write nonterminating programs. On the other hand, normalization proofs are some of the most elegant mathematics to be found in the type theory literature.

Normalization is not entirely trivial to prove, since each reduction of a term can duplicate redices in subterms.

8.1 Exercise [Quick check]: Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term? □

The calculus we shall consider in this chapter is the simply typed lambda-calculus over a single base type $\mathtt{B}$. The presentation used here is an adaptation by Martin Hofmann of a technique due to Tait [Tai67].

We begin by defining, for each type $\mathtt{T}$, a set $R_{\mathtt{T}}$ of closed terms of type $\mathtt{T}$. We will write $R_{\mathtt{T}}(\mathtt{t})$ for $\mathtt{t} \in R_{\mathtt{T}}$.

8.2 Definition:

- $R_{\mathtt{B}}(\mathtt{t})$ iff $\mathtt{t}$ halts.
- $R_{\mathtt{T_1 \to T_2}}(\mathtt{t})$ iff $\mathtt{t}$ halts and, whenever $R_{\mathtt{T_1}}(\mathtt{s})$, we have $R_{\mathtt{T_2}}(\mathtt{t\ s})$. □

8.3 Lemma:

1. If $R_{\mathtt{T}}(\mathtt{t})$, then $\mathtt{t}$ halts.
2. If $\mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t'}$, then $R_{\mathtt{T}}(\mathtt{t'})$ iff $R_{\mathtt{T}}(\mathtt{t})$. □

Proof:

1. Immediate from the definition.

2. By induction on the structure of $\mathtt{T}$.

   If $\mathtt{T} = \mathtt{B}$, the result is obvious from the definition.

   If $\mathtt{T} = \mathtt{T_1 \to T_2}$, the first requirement ($\mathtt{t}$ halts iff $\mathtt{t'}$ does) is obvious. To show the second part—that $R_T(\mathtt{t})$ iff $R_T(\mathtt{t'})$—there are two implications to establish. For the first ($\Longrightarrow$) suppose that $R_T(\mathtt{t})$ and that $R_{T_1}(\mathtt{s})$ for some $\mathtt{s : T_1}$. By definition we have $R_{T_2}(\mathtt{t\ s})$. But $\mathtt{t\ s} \longrightarrow \mathtt{t'\ s}$, from which the induction hypothesis gives us $R_{T_2}(\mathtt{t'\ s})$. Since this holds for an arbitrary $\mathtt{s}$, we have shown $R_T(\mathtt{t})$. The argument for the other direction is analogous. □

8.4 Lemma: If $\mathtt{x_1{:}T_1,\ \ldots,\ x_n{:}T_n} \vdash \mathtt{t : T}$ and $\mathtt{v_1} \ldots, \mathtt{v_n}$ are closed values with $R_{T_i}(\mathtt{v_i})$ for each i, then $R_T(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t})$. □

Proof: By induction on a derivation of $\mathtt{x_1{:}T_1,\ \ldots,\ x_n{:}T_n} \vdash \mathtt{t : T}$. The most interesting case is the one for abstraction.

Case T-VAR: $\mathtt{t} = \mathtt{x_i}$ $\quad \mathtt{T} = \mathtt{T_i}$

Immediate.

Case T-ABS: $\mathtt{t} = \lambda\mathtt{x{:}S_1.s_2}$
$\mathtt{x_1{:}T_1,\ \ldots,x_n{:}T_n,x{:}S_1} \vdash \mathtt{s_2 : S_2}$
$\mathtt{T} = \mathtt{S_1 \to S_2}$

Obviously, $\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t}$ evaluates to a value, since it is a value already. What remains to show is that $R_{S_2}((\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t})\mathtt{s})$ for any $\mathtt{s : S_1}$ such that $R_{S_1}(\mathtt{s})$. Suppose $\mathtt{s}$ is such a term. Then, by Lemma 8.3(1), we have $\mathtt{s} \longrightarrow^* \mathtt{v}$ for some $\mathtt{v}$. By Lemma 8.3(2), $R_{S_1}\mathtt{v}$. Now, by the induction hypothesis, $R_{S_2}(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\{\mathtt{x} \mapsto \mathtt{v}\}\mathtt{s_2})$. But

$$\begin{array}{ll} & (\lambda\mathtt{x{:}S_1.}\ \{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{s_2})\ \mathtt{s} \\ \longrightarrow^* & \{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\{\mathtt{x} \mapsto \mathtt{v}\}\mathtt{s_2}, \end{array}$$

from which Lemma 8.3(2) gives us

$$R_{S_2}((\lambda\mathtt{x{:}S_1.}\ \{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{s_2})\ \mathtt{s}),$$

that is,

$$R_{S_2}(((\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}(\lambda\mathtt{x{:}S_1.\ s_2}))\ \mathtt{s}).$$

Since $\mathtt{s}$ was chosen arbitrarily, the definition of $R_{S_1 \to S_2}$ gives us

$$R_{S_1 \to S_2}(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}(\lambda\mathtt{x{:}S_1.\ s_2}),$$

as required.

Case T-APP: $\mathtt{t} = \mathtt{t_1\ t_2}$
$\mathtt{x_1{:}T_1,\ \ldots,\ x_n{:}T_n} \vdash \mathtt{t_1 : T_{11}{\to}T_{12}}$
$\mathtt{x_1{:}T_1,\ \ldots,\ x_n{:}T_n} \vdash \mathtt{t_2 : T_1}$
$\mathtt{T} = \mathtt{T_{12}}$

The induction hypothesis gives us

$$R_{\mathtt{T_{11}\to T_{12}}}(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t_1})$$

and

$$R_{\mathtt{T_{11}}}(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t_2}).$$

By the definition of $R_{\mathtt{T_{11}\to T_{12}}}$,

$$R_{\mathtt{T_{12}}}((\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t_1})\ (\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}\mathtt{t_2})),$$

i.e.,

$$R_{\mathtt{T_{12}}}(\{\mathtt{x_1} \mapsto \mathtt{v_1}\}\cdots\{\mathtt{x_n} \mapsto \mathtt{v_n}\}(\mathtt{t_1\ t_2})),$$

as requred. □

8.5 Corollary [Normalization]: If $\vdash \mathtt{t : T}$, then $\mathtt{t} \longrightarrow^* \mathtt{v}$ for some $\mathtt{v}$. □

Proof: By Lemma 8.4, $R_{\mathtt{T}}\mathtt{t}$. By Lemma 8.3(1), $\mathtt{t} \longrightarrow^* \mathtt{v}$ for some $\mathtt{v}$. □

8.6 Exercise [Homework]: *(Due on Feb. 24.)* Show how to extend this proof technique to show that the simply typed lambda-calculus remains normalizing when extended with $\Leftarrow$

1. booleans
2. products. □

Chapter 9

Extensions

Explain the design principle of conservative extension.

9.1 Base Types

Base types $\rightarrow$ `B`

New syntactic forms

`T`	::=	...	*types*
		`B`	*base type*

9.2 Unit type

Unit type $\rightarrow$ `Unit`

New syntactic forms

`t`	::=	...	*terms*
		`unit`	*constant* `unit`
`T`	::=	...	*types*
		`Unit`	*unit type*

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

Comment: C and Java's `void` type can be seen as a misnomer—what they really meant is `Unit`. (The name `void` invites us to think of an empty set, which is clearly not what's meant by `void` in C-like languages, since there *are* plenty of terms that have type `void`. Saying that these terms have type `Unit` invites us instead to think that their return value is not interesting.)

9.3 Coercions

9.4 Let bindings

Let binding $\rightarrow$ `let`

New syntactic forms

`t ::=` ...		*terms*
	`let x=t in t`	*let binding*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{let x=v}_1\texttt{ in t}_2 \longrightarrow \{\texttt{x} \mapsto \texttt{v}_1\}\texttt{t}_2 \qquad \text{(E-LETBETA)}$$

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

New typing rules $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

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

9.4.1 Exercise [Homework]: *(Due on Feb 17.)* Add `let` bindings to the "`simplebool`" typechecker implementation provided on the course web page. Solution on page 286. □ ⇐

9.5 Products

- introduce new terms (pairing $(\mathtt{t}_1,\mathtt{t}_2)$ and projection $\mathtt{t.1}$ and $\mathtt{t.2}$) and product type $\mathtt{T}_1\times\mathtt{T}_2$
- new form of values $(\mathtt{v}_1,\mathtt{v}_2)$ needed
- evaluation rules
- typing rules

9.6 Records and Tuples

To obtain tuples as a special case of records, we just say that the set of field labels includes both alphabetic identifiers and natural numbers. When a field label is omitted in a record type or term, we use as the label the index where that field appears. For example, we consider `{Bool,Bool,Bool}` as an abbreviation for `{1:Bool,2:Bool,3:Bool}`. (This convention actually allows us to mix named and positional fields, writing `{a:Bool,Bool,Bool,c:Bool}` as an abbreviation for `{a:Bool,2:Bool,3:Bool,c:Bool}`, for example.)

Records and tuples $\rightarrow$ {}

New syntactic forms

$\mathtt{t}$	::=	...	*terms*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values*
		$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types*
		$\{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-RCDBETA)}$$

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

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

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

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

$$\frac{\Gamma \vdash \texttt{t}_0 : \{\texttt{l}_i\texttt{:T}_i{}^{i\in 1..n}\}}{\Gamma \vdash \texttt{t}_0\texttt{.l}_j : \texttt{T}_j} \qquad \text{(T-PROJ)}$$

New abbreviations

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

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

Note that n ranges over all nonnegative numbers here, so that records can be empty. We will assume that no record term or type contains any repeated labels.

9.6.1 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:

Record patterns → {} let *pat* **(untyped)**

New syntactic forms

p	::=	...	
		x	*variable pattern*
		$\{\texttt{l}_i\texttt{=p}_i{}^{i\in 1..n}\}$	*record pattern*
t	::=	...	*terms*

`let p=t in t` *pattern binding*

Matching rules:

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

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

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$\mathtt{let\ p=v_1\ in\ t_2} \longrightarrow match(\mathtt{p},\ \mathtt{v_1})\,\mathtt{t_2} \tag{E-LETBETA}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{let\ p=t_1\ in\ t_2} \longrightarrow \mathtt{let\ p=t_1'\ in\ t_2}} \tag{E-LET}$$

New abbreviations

$$\mathtt{\{\ldots p_i \ldots\}} \quad \stackrel{\text{def}}{=} \quad \mathtt{\{\ldots i\texttt{=}p_i \ldots\}}$$

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 type preservation and progress 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 287. □

9.7 Variants

Variants → <>

New syntactic forms

`t ::=`	...	*terms*
	`<l=t> as T`	*tagging*
	`case t of ...	l_i=x⇒t_i	...`	*case*
`T ::=`	...	*types*

$\texttt{<l}_i\texttt{:T}_i{}^{i\in 1..n}\texttt{>}$ *type of variants*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{case <l}_i\texttt{=v}_i\texttt{> as T of }\ldots\texttt{|l}_i\texttt{=x}\Rightarrow\texttt{t}_i\texttt{|}\ldots \longrightarrow \{\texttt{x} \mapsto \texttt{v}_i\}\texttt{t}_i \quad \text{(E-CASEBETA)}$$

$$\frac{\texttt{s}_1 \longrightarrow \texttt{s}_1'}{\texttt{case s}_1\texttt{ of }\ldots\texttt{|l}_i\texttt{=x}\Rightarrow\texttt{t}_i\texttt{|}\ldots \longrightarrow \texttt{case s}_1'\texttt{ of }\ldots\texttt{|l}_i\texttt{=x}\Rightarrow\texttt{t}_i\texttt{|}\ldots} \quad \text{(E-CASE)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\texttt{<l}_i\texttt{=t}_i\texttt{> as T} \longrightarrow \texttt{<l}_i\texttt{=t}_i'\texttt{> as T}} \quad \text{(E-TAG)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

$$\frac{\Gamma \vdash \texttt{t}_j \texttt{ : T}_j}{\Gamma \vdash \texttt{<l}_j\texttt{=t}_j\texttt{> as <l}_i\texttt{:T}_i{}^{i\in 1..n}\texttt{> : <l}_i\texttt{:T}_i{}^{i\in 1..n}\texttt{>}} \quad \text{(T-VARIANT)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 \texttt{ : <l}_i\texttt{:T}_i{}^{i\in 1..n}\texttt{>} \qquad \text{for each } i \quad \Gamma, \texttt{x}_i\texttt{:T}_i \vdash \texttt{t}_i \texttt{ : T}}{\Gamma \vdash \texttt{case t}_0\texttt{ of }\ldots\texttt{|l}_i\texttt{=x}_i\Rightarrow\texttt{t}_i\texttt{|}\ldots\texttt{ : T}} \quad \text{(T-CASE)}$$

Note that we syntactically allow the type label to be a non-variant type, but the typing rule then excludes this case. This is just for brevity (and implementation flexibility, e.g. when we get to definitions).

9.8 Sums

9.9 General Recursion

General recursion $\rightarrow$ `fix`

New syntactic forms

`t ::=`	...	*terms*
	`fix t`	*fixed point of t*

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : T}_1\texttt{→T}_1}{\Gamma \vdash \texttt{fix t}_1 \texttt{ : T}_1} \quad \text{(T-FIX)}$$

New abbreviations

$$\texttt{letrec x=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{let x = fix (}\lambda\texttt{x.t}_1\texttt{) in t}_2$$

A corollary of the definability of fixed-point combinators at every type is that every type in this system is inhabited. For example, for each type `T`, the application `(diverge`$_T$ `unit)` has type `T`, where `diverge`$_T$ is defined like this:

```
divergeT = λ_:Unit. fix (λx:T. x);
▶ T = Nat: *
divergeT : Unit → T
```

9.10 Lists

Lists → 𝔹 `List`

New syntactic forms

`t ::=`	...	*terms*
	`nil [T]`	*empty list*
	`cons [T] t t`	*list constructor*
	`null [T] t`	*test for empty list*
	`head [T] t`	*head of a list*
	`tail [T] t`	*tail of a list*
`v ::=`	...	*values*
	`nil [T]`	*empty list*
	`cons [T] v v`	*list constructor*
`T ::=`	...	*types*
	`List T`	*type of lists*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{cons[T]}\ \mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{cons[T]}\ \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-CONS1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{cons[T]}\ \mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{cons[T]}\ \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-CONS2)}$$

$$\mathtt{null[S]}\ \mathtt{(nil[T])} \longrightarrow \mathtt{true} \quad \text{(E-NULLBETAT)}$$

$$\mathtt{null[S]}\ \mathtt{(cons[T]}\ \mathtt{v}_1\ \mathtt{v}_2\mathtt{)} \longrightarrow \mathtt{false} \quad \text{(E-NULLBETAF)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{null[T]}\ \mathtt{t}_1 \longrightarrow \mathtt{null[T]}\ \mathtt{t}_1'} \quad \text{(E-NULL)}$$

$$\texttt{head[S] (cons[T] v}_1\texttt{ v}_2\texttt{)} \longrightarrow \texttt{v}_1 \qquad \text{(E-HeadBeta)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{head[T] t}_1 \longrightarrow \texttt{head[T] t}_1'} \qquad \text{(E-Head)}$$

$$\texttt{tail[S] (cons[T] v}_1\texttt{ v}_2\texttt{)} \longrightarrow \texttt{v}_2 \qquad \text{(E-TailBeta)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{tail[T] t}_1 \longrightarrow \texttt{tail[T] t}_1'} \qquad \text{(E-Tail)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

$$\Gamma \vdash \texttt{nil [T] : List T} \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : T}_1 \qquad \Gamma \vdash \texttt{t}_2 \texttt{ : List T}_1}{\Gamma \vdash \texttt{cons[T}_1\texttt{] t}_1\texttt{ t}_2 \texttt{ : List T}_1} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : List T}_1}{\Gamma \vdash \texttt{null[T}_1\texttt{] t}_1 \texttt{ : Bool}} \qquad \text{(T-Null)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : List T}_1}{\Gamma \vdash \texttt{head[T}_1\texttt{] t}_1 \texttt{ : T}_1} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : List T}_1}{\Gamma \vdash \texttt{tail[T}_1\texttt{] t}_1 \texttt{ : List T}_1} \qquad \text{(T-Tail)}$$

9.11 Lazy records and let-bindings

Lazy let bindings $\rightarrow$ `let` `lazy`

New syntactic forms

`t ::=`	...	*terms*
	`lazy let x=t in t`	*lazy let binding*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{lazy let x=t}_1\texttt{ in t}_2 \longrightarrow \{\texttt{x} \mapsto \texttt{t}_1\}\texttt{t}_2 \qquad \text{(E-LLetBeta)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

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

Lazy records $\to$ {} `lazy`

New syntactic forms

$\mathtt{t}$ ::=	...	*terms*
	$\mathtt{lazy\ \{l_i{=}t_i}^{\ i\in 1..n}\mathtt{\}}$	*lazy record*
$\mathtt{v}$ ::=	...	*values*
	$\mathtt{lazy\ \{l_i{=}t_i}^{\ i\in 1..n}\mathtt{\}}$	*lazy record value*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{(lazy\ \{l_i{=}t_i}^{\ i\in 1..n}\mathtt{\}).l_i} \longrightarrow \mathtt{t_i} \qquad \text{(E-LRcdBeta)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash^{\ i\in 1..n}\ \mathtt{t_i} : \mathtt{T_i}}{\Gamma \vdash \mathtt{lazy\ \{l_i{=}t_i}^{\ i\in 1..n}\mathtt{\}} : \mathtt{\{l_i{:}T_i}^{\ i\in 1..n}\mathtt{\}}} \qquad \text{(T-LRcd)}$$

Show how to achieve mutual recursion with fixedpoints of records.

Chapter 10

Exceptions

So far, we have considered a variety of features found in "pure" functional programming languages, including functional abstraction, basic types such as numbers and booleans, and structured types such as records and variants. These features form the backbone of essentially all modern programming languages. Most real languages, though, are much richer semantically than the simple calculi we have seen: besides yielding results, evaluation of expressions may engender a variety of **computational effects**:

- input and output
- assignment to mutable variables (reference cells, arrays, mutable data structures, etc.)
- non-local transfer of control via exceptions, continuations, and other control operators
- inter-process communication.

In this chapter and the next, we'll see how various sorts of effects of can be incorporated in the models we have developed. We begin with exception handling—a simple form of control operator. We'll examine exceptions first in a very simple form, then see how it can be extended to more realistic mechanisms.

10.1 Simple Exceptions

10.2 Handling Exceptions

Errors $\rightarrow$ *error*

New syntactic forms

$\mathtt{t}$ ::= ...		*terms*
	$\mathtt{error}$	*run-time error*
	$\mathtt{try}\ \mathtt{t}_1\ \mathtt{otherwise}\ \mathtt{t}_2$	*trap errors*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{try}\ \mathtt{v}_1\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1 \qquad \text{(E-TryBetaOK)}$$

$$\mathtt{try}\ \mathtt{error}\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{t}_2 \qquad \text{(E-TryBetaErr)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{try}\ \mathtt{t}_1\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{try}\ \mathtt{t}_1'\ \mathtt{otherwise}\ \mathtt{t}_2} \qquad \text{(E-Try)}$$

$$\mathtt{if}\ \mathtt{error}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{error} \qquad \text{(E-IfErr)}$$

$$\mathtt{error}\ \mathtt{t}_2 \longrightarrow \mathtt{error} \qquad \text{(E-AppErr1)}$$

$$\mathtt{v}_1\ \mathtt{error} \longrightarrow \mathtt{error} \qquad \text{(E-AppErr2)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\Gamma \vdash \mathtt{error} : \mathtt{T} \qquad \text{(T-Error)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{try}\ \mathtt{t}_1\ \mathtt{otherwise}\ \mathtt{t}_2 : \mathtt{T}} \qquad \text{(T-Try)}$$

10.2.1 Theorem [Preservation]: Unchanged. □

10.2.2 Theorem [Progress]: Suppose $\mathtt{t}$ a closed normal form. If $\vdash \mathtt{t} : \mathtt{T}$, then either $\mathtt{t}$ is a value or $\mathtt{t} = \mathtt{error}$. □

10.2.3 Exercise [Homework]: *(Due on Feb. 22.)* $\Leftarrow$

1. Show that the rules defining the typing relation here do not lead immediately to an algorithm for typechecking.

2. Show that the typing relation is still decidable by writing down an algorithm that, given Γ and `t`, either finds some `T` such that $\Gamma \vdash \mathtt{t} : \mathtt{T}$ or else determines that there is no such `T`. □

10.3 Exceptions Carrying Values

First, we fix a type $\mathtt{T}_{exn}$.

Exceptions $\rightarrow$ *exceptions*

New syntactic forms

`t ::=`	...	*terms*
	`raise t`	*raise exception*
	`try t`$_1$ `otherwise t`$_2$	*trap exceptions*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{try}\ \mathtt{v}_1\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1 \quad \text{(E-TryBetaOK)}$$

$$\mathtt{try}\ \mathtt{raise}\ \mathtt{v}_{11}\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{t}_2\ \mathtt{v}_{11} \quad \text{(E-TryBetaExn)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{try}\ \mathtt{t}_1\ \mathtt{otherwise}\ \mathtt{t}_2 \longrightarrow \mathtt{try}\ \mathtt{t}_1'\ \mathtt{otherwise}\ \mathtt{t}_2} \quad \text{(E-Try)}$$

$$\mathtt{if}\ \mathtt{raise}\ \mathtt{v}_{11}\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \quad \text{(E-IfExn)}$$

$$(\mathtt{raise}\ \mathtt{v}_{11})\ \mathtt{t}_2 \longrightarrow \mathtt{raise}\ \mathtt{v}_{11} \quad \text{(E-AppExn1)}$$

$$\mathtt{v}_1\ (\mathtt{raise}\ \mathtt{v}_{21}) \longrightarrow \mathtt{v}_{21} \quad \text{(E-AppExn2)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{exn}}{\Gamma \vdash \mathtt{raise}\ \mathtt{t}_1 : \mathtt{T}} \quad \text{(T-Exn)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{exn} \rightarrow \mathtt{T}}{\Gamma \vdash \mathtt{try}\ \mathtt{t}_1\ \mathtt{otherwise}\ \mathtt{t}_2 : \mathtt{T}} \quad \text{(T-Try)}$$

Choices for $\mathtt{T}_{exn}$:

- $\mathtt{T}_{exn} = \mathtt{Nat}$. Unix `errno` convention.

- $\mathtt{T}_{exn}$ = `String`. Similar, but errors can be mapped to more descriptive strings. (Requires error handlers to *parse* messages to distinguish between errors.)
- $\mathtt{T}_{exn}$ is some variant type. Better, but inflexible.
- $\mathtt{T}_{exn}$ is an *extensible type*. In ML, this type is called `exn`. Writing an exception declaration effectively adds a new clause to a global datatype definition for `exn`. A handler effectively pattern matches against elements of this type. So we can write, approximately:

```
raise E(t)              ≝  raise <E=t>
try t with E(x) → h     ≝  try t otherwise
                             λe. case e of <E=x>→h | _ → raise e
```

10.4 Further Reading

The exposition in this chapter is adapted from a lecture by Robert Harper.

Chapter 11

References

Outline:

- Treating mutable reference cells requires an extension of the framework we've developed so far: we need to be able to talk explicitly about the *heap*. This extension is fairly straightforward; the most interesting part is how we need to change the statement of the type preservation theorem.

- Intro

  - References are mutable cells holding values
  - Introduces the notion of a **heap** or **store**
  - Basic operations: allocation, dereferencing, assignment
  - Critical issue: **aliasing**. Show several examples. Aliasing analysis is very hard! (Cite Reynolds' type systems for aliasing, and their successors.)
  - Note that in many programming languages every variable is implicitly a reference cell: wherever you see `T`, you should actually read `Ref(T)`. (Strictly speaking, we should even read `Ref(Option(T))`, to take `null` into account.) Whether references should be implicit or explicit is very much a matter of taste. Implicit is a bit more concise, but encourages use of mutation where it is not really needed, making programs harder to reason about.

- Syntax

  - Add standard (ML-style) operations for creating reference cells, assigning, and dereferencing.
  - (We can also add pointer equality if we like. Won't bother to do it here.)

- Typing rules for these constructs are obvious. (Show them.)
- Operational semantics:
  - What are the values of type `Ref(T)`?
    - Naive answer: terms of the form `ref v`
    - But this doesn't give us what we expect: executing a `ref` should actually *allocate* storage. For example, `(ref 0, ref 0)` does not mean the same as `(λx:Ref(Int). (x,x)) (ref 0)`: the former allocates two reference cells, while the latter allocates just one and builds a pair containing two pointers to it.
    - If evaluating `ref 0` allocates some storage, then its result must be a **pointer** to that storage.
  - Evidently we need to add pointers or **locations** (maybe they should just be called pointers??) to the set of values.
  - Carry along a *store* modeling the heap. A program now is not just a term, it is a term plus a store. The evaluation relation maps programs to programs.
  - Existing evaluation rules are straightforwardly augmented with stores.
  - Add new evaluation rules for new constructs.
- Typing again
  - Since we added them to values, we also need to add locations to the term syntax. Of course, we don't expect the programmer to write programs with locations in them—this is just a notational trick that allows us to keep using our standard operational semantics.
  - What is the type of a program containing locations? Depends on the types of what's stored in the locations!
  - So a first try for the typing rule for locations could be:

    $$\frac{\Gamma \vdash \mu(l) : T}{\Gamma \vdash l : T}$$

    This rule depends on the current store, so we'd need to augment the context (in every typing rule) with a store:

    $$\frac{\Gamma \mid \mu \vdash \mu(l) : T}{\Gamma \mid \mu \vdash l : T}$$

    Show an example or two of typing derivations involving typechecking the store.

  - But this is not very efficient, since we might have to keep typechecking the whole store over and over every time a location is mentioned in the program. Moreover, it doesn't work, because the store itself may be circular. (Show an example of a circular store. Exercise: write a program that produces such a store.) In fact, there may not even be a unique typing for such a store.
  - But notice that, whenever we allocate a location in the store, we always have a type in mind for the value that's going to be stored there. Adding all these together, we can speak of a **store typing** that associates each location with its intended type.
  - It is easy to check whether a store (even a circular one!) is consistent with a given store typing.
  - Existing typing rules are straightforwardly augmented with store typings.
  - Only interesting rule is T-LOC, which is analogous to T-VAR.

- Soundness
  - Preservation theorem now needs to say something about how the store and store typing are affected by a step of evaluation.
  - First try just adds a store and a store typing to the premises of the theorem. But this doesn't give us enough information to know that the preservation theorem applies again on the next step of evaluation: we need to extend the conclusions as well, to show that the new store is well typed against the store typing.
  - This version works for the deref and assignment rules, but it doesn't handle the allocation rule, which changes not only the values in the store but the size of the store's domain. To match this new domain, we need a bigger store typing.
  - Final version of the theorem says that there is some new store typing that agrees with the old one everywhere (this condition is not actually needed, is it?) and such that the new store is typed under the new typing. This is strong enough to "turn the crank" repeatedly.

References $\rightarrow$ `Unit` `:=`

Syntax

`t ::=`		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`unit`	*constant* `unit`
	`l`	*store location*
	`ref t`	*reference creation*
	`!t`	*dereference*
	`t:=t`	*assignment*
`v ::=`		*values*
	`λx:T.t`	*abstraction value*
	`l`	*store location*
`T ::=`		*types*
	`T→T`	*type of functions*
	`Unit`	*unit type*
	`Ref` $\mathtt{T_1}$	*type of reference cells*
Γ `::=`		*contexts*
	$\emptyset$	*empty context*
	Γ`, x:T`	*term variable binding*
μ `::=`		*stores*
	$\emptyset$	*empty store*
	$\mu, \mathtt{l} = \mathtt{v}$	*location binding*
Σ `::=`		*store typings*
	$\emptyset$	*empty store typing*
	Σ`, l:T`	*location typing*

Evaluation $\boxed{\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'}$

$$(\lambda \mathtt{x}\mathtt{:T_{11}}.\mathtt{t_{12}})\ \mathtt{v_2} \mid \mu \longrightarrow \{\mathtt{x} \mapsto \mathtt{v_2}\}\mathtt{t_{12}} \mid \mu \qquad \text{(E-BETA)}$$

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

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

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

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\texttt{ref } t_1 \mid \mu \longrightarrow \texttt{ref } t_1' \mid \mu'} \qquad \text{(E-REF1)}$$

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

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(E-DEREF1)}$$

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

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t_1' := t_2 \mid \mu'} \qquad \text{(E-ASSIGN1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t_2' \mid \mu'} \qquad \text{(E-ASSIGN2)}$$

Typing $\boxed{\Gamma \mid \Sigma \vdash t : T}$

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

$$\frac{\Gamma, x{:}T_1 \mid \Sigma \vdash t_2 : T_2}{\Gamma \mid \Sigma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

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

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

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \texttt{ref } t_1 : \texttt{Ref } T_1} \qquad \text{(T-REF)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Ref } T_1}{\Gamma \mid \Sigma \vdash !t_1 : T_1} \qquad \text{(T-DEREF)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Ref } T_1 \qquad \Gamma \mid \Sigma \vdash t_2 : T_1}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \texttt{Unit}} \qquad \text{(T-ASSIGN)}$$

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

11.2 Exercise: Can you find a context Γ, a store μ, and two different store typings Σ_1 and Σ_2 such that both $\Gamma \mid \Sigma \vdash \mu_1$ and $\Gamma \mid \Sigma \vdash \mu_2$? Solution on page 287. □

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

Proof: Just like the proof of Lemma 7.4.8. □

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

Proof: Easy. □

11.5 Theorem [Preservation]: If

$$\begin{array}{l} \Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T} \\ \Gamma \mid \Sigma \vdash \mu \\ \mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu' \end{array}$$

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

$$\begin{array}{l} \Gamma \mid \Sigma' \vdash \mathtt{t}' : \mathtt{T} \\ \Gamma \mid \Sigma' \vdash \mu'. \end{array}$$

□

Proof: Straightforward induction on typing derivations, using the lemmas above and the evident inversion property of the typing rules. □

11.6 Exercise [Homework]: *(Due on Feb. 29.)* Is the evaluation relation defined in this chapter strongly normalizing on well-typed terms? If so, prove it. If not, show how to write a well-typed factorial function in the present calculus extended with numbers and booleans. Solution on page 288. □ $\Leftarrow$

11.1 Type and Effect Systems

Quick sketch of region typing.

11.2 Further Reading

The presentation in this chapter is adapted from Harper's treatment [Har94, Har96]. An earlier account in a similar style is given by Wright and Felleisen [WF94].

The combination of references (or other computational effects) with ML-style polymorphic type inference raises some subtle problems (cf. Section **??**) 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 12

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, consider the rule T-APP for typing applications:

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}} \to \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \qquad \text{(T-APP)}$$

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**:

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

No other changes are needed to the typing relation (though, as we shall see in Section 12.2, 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 λ<:$_{\{\}}$, 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.

12.1 The Subtype Relation

For record types, we have already seen that we want to consider the type $S = \{k_1{:}S_1 \ldots k_m{:}S_m\}$ to be a subtype of $T = \{l_1{:}T_1 \ldots l_n{:}T_n\}$ if T is obtained by dropping fields from S.

$$\{l_i{:}T_i\ ^{i\in 1..n+k}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\} \qquad \text{(S-RCD-WIDTH)}$$

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 S are subtypes of their types in T.

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i{:}S_i\ ^{i\in 1..n}\} <: \{l_i{:}T_i\ ^{i\in 1..n}\}} \qquad \text{(S-RCD-DEPTH)}$$

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.

$$\frac{\pi \text{ is a permutation of } \{1..n\}}{\{l_i{:}T_i\ ^{i\in 1..n}\} <: \{l_{\pi i}{:}T_{\pi i}^{\ i\in 1..n}\}} \qquad \text{(S-RCD-PERM)}$$

For the types of functions, subtyping is a little trickier. 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 (T_1 <: S_1) and none of the results that it returns will surprise the context (S_2 <: T_2).

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

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.

12.1.1 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. □

Next, 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.

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

(The Top type is an optional part of the system: everything makes sense without it.)

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

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

and second, that it should be transitive:

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

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

Formally, the subtype relation is the least relation closed under the rules we have given. For easy reference, we recapitulate the full definition of the calculus, highlighting the syntax and rules we have just added.

Simply typed lambda calculus with records and subtyping $\rightarrow$ {} <:

Syntax

t ::=		*terms*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	$\{l_i = t_i\ ^{i \in 1..n}\}$	*record*
	t.l	*projection*
v ::=		*values*
	λx:T.t	*abstraction value*
	$\{l_i = v_i\ ^{i \in 1..n}\}$	*record value*
T ::=		*types*
	Top	*maximum type*
	T$\rightarrow$T	*type of functions*
	$\{l_i : T_i\ ^{i \in 1..n}\}$	*type of records*
Γ ::=		*contexts*
	$\emptyset$	*empty context*
	Γ, x:T	*term variable binding*

Evaluation $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

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

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-APP2)}$$

$$\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-RCDBETA)}$$

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

$$\frac{\begin{array}{c}\mathtt{t}_j \\ \longrightarrow \mathtt{t}_j'\end{array}}{\{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=}\mathtt{t}_j\mathtt{,l}_k\mathtt{=}\mathtt{t}_k\ ^{k\in j+1..n}\} \longrightarrow \{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=}\mathtt{t}_j'\mathtt{,l}_k\mathtt{=}\mathtt{t}_k\ ^{k\in j+1..n}\}}$$

(E-RECORD)

Subtyping $\boxed{\mathtt{S} \mathrel{<:} \mathtt{T}}$

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

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

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

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

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

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

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

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

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

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

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

$$\frac{\Gamma \vdash \mathtt{t}_0 : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i{}^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_0\mathtt{.l}_j : \mathtt{T}_j} \quad \text{(T-PROJ)}$$

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

12.1.2 Exercise: In what ways are the `Top` type and the empty record type `{}` similar? In what ways are they different? Solution on page 288. □

12.1.3 Exercise:

1. How many different supertypes does the type `{a:Top,b:Top}` have?
2. Can you find an infinite descending chain in the subtype relation—that is, an infinite sequence of types $\mathtt{S}_0$, $\mathtt{S}_1$, etc. such that each $\mathtt{S}_{i+1}$ is a subtype of $\mathtt{S}_i$?
3. What about an infinite ascending chain?

Solution on page 288. □

12.1.4 Exercise:

1. Is there a type that is a subtype of every other type?
2. Is there an arrow type $\mathtt{T}_1 \to \mathtt{T}_2$ that is a supertype of every other arrow type $\mathtt{S}_1 \to \mathtt{S}_2$?

Solution on page 289. □

12.1.5 Exercise: Suppose we extend the calculus with the product type constructor $\mathtt{T}_1 \times \mathtt{T}_2$ described in Section 9.5. It is natural to add a subtyping rule

$$\frac{\mathtt{S}_1 <: \mathtt{T}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \times \mathtt{S}_2 <: \mathtt{T}_1 \times \mathtt{T}_2} \quad \text{(S-PROD)}$$

corresponding to the depth subtyping rule S-RCD-DEPTH for records. Would it be a good idea to add a width subtyping rule for products

$$\mathtt{T}_1 \times \mathtt{T}_2 <: \mathtt{T}_1 \quad \text{(S-PROD-WIDTH)}$$

as well? Solution on page 289. □

12.2 Safety

12.3 Variance

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

$$\mathtt{T}[\mathtt{\{\}}] = \mathtt{\{x:Top,y:\{\}\}}\to\mathtt{Top}\to\mathtt{\{\}}\to\mathtt{\{\}}.$$

We say that $\mathtt{T}[\,]$ is **covariant** in the position(s) marked by the holes if, whenever $\mathtt{S}_1\mathtt{<:}\mathtt{S}_2$, we have $\mathtt{T}[\mathtt{S}_1]$ <: $\mathtt{T}[\mathtt{S}_2]$. We say that $\mathtt{T}[\,]$ is **contravariant** if, whenever $\mathtt{S}_1\mathtt{<:}\mathtt{S}_2$, we have $\mathtt{T}[\mathtt{S}_2]$ <: $\mathtt{T}[\mathtt{S}_1]$. If $\mathtt{T}[\,]$ 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 $[\,]\to\mathtt{S}$ is contravariant while $\mathtt{S}\to[\,]$ is covariant.

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

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

12.4 Coercion Semantics

12.5 Subtyping and References

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

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

for subtyping between reference cell types? □

12.6 Primitive Subtyping

12.7 The Bottom Type

Chapter 13

Implementation of Subtyping

In this chapter we consider the algorithmic properties of the simply typed lambda-calculus with records and subtyping.

13.1 Algorithmic Subtyping

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

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

13.1.1 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:

13.1.2 Lemma:

1. $\mathtt{S} \mathrel{<:} \mathtt{S}$ can be derived for every type $\mathtt{S}$ without using S-REFL.

2. If $\mathtt{S} \mathrel{<:} \mathtt{T}$ can be derived at all, then it can be derived without using S-TRANS. □

Proof: The first part goes by induction on the structure of $\mathtt{S}$, the second by induction on the given subtyping derivation. □

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

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

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

$$\frac{\mapsto \mathtt{T_1} <: \mathtt{S_1} \qquad \mapsto \mathtt{S_2} <: \mathtt{T_2}}{\mapsto \mathtt{S_1 \to S_2} <: \mathtt{T_1 \to T_2}} \qquad \text{(SA-ARROW)}$$

$$\frac{\{\mathtt{l_i}\ ^{i \in 1..n}\} \subseteq \{\mathtt{k_j}\ ^{j \in 1..m}\} \qquad \text{if } \mathtt{k_j} = \mathtt{l_i}\text{, then} \quad \mapsto \mathtt{S_j} <: \mathtt{T_i}}{\mapsto \{\mathtt{k_j}\mathtt{:}\mathtt{S_j}\ ^{j \in 1..m}\} <: \{\mathtt{l_i}\mathtt{:}\mathtt{T_i}\ ^{i \in 1..n}\}} \qquad \text{(SA-RCD)}$$

□

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

Proof: By induction on derivations, 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 13.1.5). The realization of this algorithm in ML appears in Section 13.3.

13.2 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 $\Gamma \mapsto \mathtt{t} : \mathtt{T}$. 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 $\Gamma \mapsto \mathtt{t} : \mathtt{T}$ 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:

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

Any such derivation can be rearranged like this:

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

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

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t_1 : S_1\to S_2} \qquad \dfrac{\dfrac{\vdots}{T_1 <: S_1} \qquad \dfrac{\vdots}{S_2 <: T_2}}{S_1\to S_2 <: T_1\to T_2}\;(\text{S-Arrow})}{\Gamma \vdash t_1 : T_1\to T_2}\;(\text{T-Sub}) \qquad \dfrac{\vdots}{\Gamma \vdash t_2 : T_1}}{\Gamma \vdash t_1\ t_2 : T_2}$$

can be replaced by one of the form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t_1 : S_1\to S_2} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash t_2 : T_1} \qquad \dfrac{\vdots}{T_1 <: S_1}}{\Gamma \vdash t_2 : S_1}\;(\text{T-Sub})}{\Gamma \vdash t_1\ t_2 : S_2}\;(\text{T-App}) \qquad \dfrac{\vdots}{S_2 <: T_2}}{\Gamma \vdash t_1\ t_2 : T_2}\;(\text{T-Sub})$$

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

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t} : \mathtt{S}} \qquad \dfrac{\vdots}{\mathtt{S} <: \mathtt{U}}}{\Gamma \vdash \mathtt{t} : \mathtt{U}}\;(\text{T-SUB}) \qquad \dfrac{\vdots}{\mathtt{U} <: \mathtt{T}}}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\;(\text{T-SUB})$$

can be rewritten:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t} : \mathtt{S}} \qquad \dfrac{\dfrac{\vdots}{\mathtt{S} <: \mathtt{U}} \qquad \dfrac{\vdots}{\mathtt{U} <: \mathtt{T}}}{\mathtt{S} <: \mathtt{T}}\;(\text{S-TRANS})}{\Gamma \vdash \mathtt{t} : \mathtt{T}}\;(\text{T-SUB})$$

13.2.1 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:

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

$$\dfrac{\mathtt{x}\!:\!\mathtt{T} \in \Gamma}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{x} : \mathtt{T}} \qquad \text{(TA-VAR)}$$

$$\dfrac{\Gamma, \mathtt{x}\!:\!\mathtt{T}_1 \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda\mathtt{x}\!:\!\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(TA-ABS)}$$

$$\dfrac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 = \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \mathrel{|\!\!\blacktriangleright} \mathtt{T}_2 <: \mathtt{T}_{11}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(TA-APP)}$$

$$\dfrac{\text{for each } i \quad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_i : \mathtt{T}_i}{\Gamma \mathrel{|\!\!\blacktriangleright} \{\mathtt{l}_1\!=\!\mathtt{t}_1 \ldots \mathtt{l}_n\!=\!\mathtt{t}_n\} : \{\mathtt{l}_1\!:\!\mathtt{T}_1 \ldots \mathtt{l}_n\!:\!\mathtt{T}_n\}} \qquad \text{(TA-RCD)}$$

$$\dfrac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t} : \{\mathtt{l}_1\!:\!\mathtt{T}_1 \ldots \mathtt{l}_n\!:\!\mathtt{T}_n\}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}.\mathtt{l}_i : \mathtt{T}_i} \qquad \text{(TA-PROJ)}$$

□

13.2.3 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.)

13.2.4 Theorem [Soundness of the algorithmic typing relation]: If $\Gamma \mapsto \texttt{t : T}$, 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 13.2.4 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.

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

Proof: Exercise (recommended). □

13.2.6 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. □

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

13.2.8 Exercise [Recommended]: Show how to extend the proof of preservation for the simply typed lambda-calculus (7.4.10) to the system with subtyping. □

13.3 Concrete Realization

Recall the datatype definitions for types and terms:

```
type ty =
    TyArr of ty * ty
  | TyTop
  | TyRecord of (string * ty) list

type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmRecord of info * (string * term) list
  | TmProj of info * term * string
```

The algorithmic subtype relation can be translated directly into OCaml as follows:

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

The only complication here arises from the subtyping rule for records, which involves a certain amount of fussing around with list operations. `List.forall` applies a predicate (its second argument) to every member of a list and returns `true` iff all these applications return `true`. `List.assoc li fS` looks up the label `li` in the list of fields `fS` and returns the associated field type `tySi`; if `li` is not among the labels in `fS`, it raises the exception `Not_found`.

The algorithm for typechecking is more straightforward.

```
let rec typeof ctx t =
  match t with
    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, tyT)
| 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")
| 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")
```

Note that the subtype checking function can be made more user-friendly by making it either succeed (returning the unit value `()`) or raise an exception, rather than returning a boolean. Notice that this doesn't change the end-to-end behavior of the checker: if the subtype checker returns false, the typechecker is always going to stop immediately at that point.

13.4 Meets and Joins

To deal with conditional expressions, we need to be able to find the smallest type that could have been given (using T-SUB) to both the `then` and the `else` branches. That is, we need to find the smallest type that lies above the minimal types of the two branches.

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

$\vdash$ S <: J
$\vdash$ T <: J
for all types U, if $\vdash$ S <: U and $\vdash$ T <: U then $\vdash$ 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

$\vdash$ M <: S
$\vdash$ M <: T
for all types L, if $\vdash$ L <: S and $\vdash$ L <: T then $\vdash$ L <: M.

A pair of types S and T is said to be **bounded below** if there is some type L such that $\vdash$ L <: S and $\vdash$ 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.

Using the `join` function we can now give an algorithmic rule for the `if` construct in the presence of subtyping.

```
let rec join tyS tyT =
  if subtype tyS tyT then tyT
  else if subtype tyT tyS then tyS
  else match (tyS,tyT) with
        (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))
      | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
           (try TyArr(meet tyS1 tyT1, join tyS2 tyT2)
            with Not_found → TyTop)
      | _ → TyTop

and meet tyS tyT =
  if subtype tyS tyT then tyS
  else if subtype tyT tyS then tyT
  else
    match (tyS,tyT) with
        (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)
       | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
            TyArr(join tyS1 tyT1, meet tyS2 tyT2)
       | _ → 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"
```

Chapter 14

Imperative Objects

This chapter needs a little more writing, but my main annoyance with it is that the final section (on "self") requires lazy evaluation at the moment. There are some fairly deep issues here (Didier and I have been discussing for years whether it is actually possible to build a completely satisfactory object model by translation to lambda-calculus, and this is one of the biggest reasons). I think the material is still valuable even so, but I'd love to find a better way around this point.

Some of the examples in this chapter would look nicer if we added operations for record extension and overriding (not polymorphic: just on ordinary records). The overriding operation should, if possible, be consistent with the polymorphic record update operation.

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.

14.1 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 11.)

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
```

14.2 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
```

14.3 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
```

14.4 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
```

14.5 Extending the Internal State

Suppose we want to define a class of `BackupCounter`s 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.

14.6 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.

14.6.1 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
```

14.6.2 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 14.3).
3. Extend this subclass with yet another subclass that supports backups as well (as in Section 14.5). □

Chapter 15

Recursive Types

the type IntList is an infinite tree

so let's just allow types to be infinite; use the μ notation as a shorthand for infinite trees

(Of course, the typechecker can't actually manipulate infinite trees; later in the chapter we will see two different ways of dealing with μ types more realistically)

15.1 Examples

Lists

Hungry Functions

As another simple illustration of the use of recursive types, here is a type of functions that can accept any number of numeric arguments:

```
Hungry = μA. Nat → A;
```

An element of this type can be defined using the least fixed point operator on values:

```
f =
  fix
    (λf: Nat→Hungry.
     λn:Nat.
       f);
```

```
▸ f : Nat → Nat → Hungry
```

```
f 0 1 2 3 4 5;
```

```
▸ (λn:Nat. fix λf':Nat→Hungry. λn':Nat.f') : μA. Nat → A
```

Recursive Values from Recursive Types

A more challenging example—and one that reveals some of the power of the extension we are making—uses recursive types to write down a well-typed version of the least-fixed-point combinator:

```
fixedpoint_T =
  λf:T→T.
    (λx:(μA.A→T). f (x x))
    (λx:(μA.A→T). f (x x));
▶ fixedpoint_T : (T→T) → T
```

Untyped Lambda-Calculus, Redux

Perhaps the best illustration of the power of recursive types is the observation that it is possible to embed the whole untyped lambda-calculus (in a well-typed way!) into a statically typed calculus with recursive types. Let `D` be the following **universal type**:

```
D = μX. X→X;
```

(Note that the form of `D`'s definition is reminiscent of the defining properties of "universal domains" in denotational semantics.) Now define an "injection function" `lam` mapping functions from `D` to `D` into elements of `D` as follows:

```
lam = λf:D→D. f;
▶ lam : (D→D) → D → D

ap = λf:D. λa:D. f a;
▶ ap : D → D → (μX. X → X)
```

Now, suppose `M` is a closed pure lambda-term involving just variables, abstractions, and applications. Then we can construct an element of `D` representing `M`, written `M*`, in a uniform way as follows:

$$
\begin{array}{lcl}
\mathtt{x}^\star & = & \mathtt{x} \\
(\lambda\mathtt{x.M})^\star & = & \mathtt{lam}\ (\lambda\mathtt{x:D.}\ \mathtt{M}^\star) \\
(\mathtt{M\ N})^\star & = & \mathtt{ap}\ \mathtt{M}^\star\ \mathtt{N}^\star
\end{array}
$$

For example, here is how the untyped fixed point combinator is expressed as an element of `D`:

```
fixD = lam (λf:D.
          ap (lam (λx:D. ap f (ap x x)))
             (lam (λx:D. ap f (ap x x))));
▶ fixD : D → D
```

15.1.1 Exercise: Note that the term defining `fixedpoint` contains many redices. Use the reduction rules (C-UNFOLDFOLD plus the usual β-reduction rule) to find its normal form. What is the type-erasure of this normal form? □

We can go even further, if we work in a language with variants as in Section 9.7. Then we can extend the datatype of pure lambda-terms to include numbers like this:

```
D = μX. <nat:Nat, fn:X→X>;
```

That is, an element of `D` is either a number or a function from `D` to `D`, tagged `nat` or `fn`, repectively. It will also be convenient to have an abbreviation for the "once-unrolled" body of `D`:

```
DBody = <nat:Nat, fn:D→D>;
```

The implementation of the `lam` constructor is essentially the same as before:

```
lam = λf:D→D.
        <fn=f> as DBody;
▶ lam : (D→D) → DBody
```

The implementation of `ap`, though, is different in an interesting way:

```
ap =
  λf:D. λa:D.
    case f of
      nat=n ⇒ diverge_D unit
    | fn=f ⇒ f a;
▶ ap : D → D → D
```

Notice how closely the tag-checking going on here resembles the run-time tag checking inside an implementation of a strongly-but-latently typed language such as Scheme [**?**]. In this sense, typed computation may be said to "include" untyped or dynamically typed computation. Similar tag checking is needed in order to define the successor function on elements of `D`:

```
suc =
  λf:D.
    case f of
      nat=n ⇒ (<nat=succ n> as DBody)
    | fn=f ⇒ diverge_D unit;
```

The injection of `0` into `D` is trivial:

```
zro = <nat=0> as DBody;
▶ suc : D → DBody
  zro : DBody
```

15.1.2 Exercise: Use the untyped fixed point combinator that we saw above to define an untyped factorial function on `D`. □

15.1.3 Exercise [Recommended]: Using the `fullrec` checker from the course web directory, extend the datatype `D` to include untyped records

```
D = μX. <nat:Nat, fn:X→X, rcd:Nat→X>;
```

and implement record construction and field projection. For simplicity, use natural numbers as field labels—i.e., records are represented as functions from natural numbers to elements of `D`. (N.b.: this exercise has nothing to do with hungry functions!) □

Recursive Objects

```
Counter = μP. {get:Nat, inc:Unit→P};

p =
  let create =
    fix
      (λcr: {x:Nat}→Counter.
        λs: {x:Nat}.
          {get = s.x,
           inc = λ_:Unit. cr {x=succ(s.x)}})
  in
    create {x=0};
```

```
▸ p : {get:Nat, inc:Unit→Counter}
```

```
p1 = p.inc unit;
p1.get;
```

```
▸ p1 : Counter
  1 : Nat
```

15.2 Equi-recursive Types

note that we cannot obtain all infinite trees by writing finite expressions using μ, just the *regular* ones.

Prove this by writing a function that expands a μ type into its regular tree form, and showing that this function is not surjective (maybe the latter is an exercise). introduce the idea of contractive types

- use the equivalence relation (should introduce it previously) to capture the intuition that a recursive type is "the same as" its unfolding

- but this doesn't give us enough equivalences. For example $\mu X.T\to T\to X$ and $T\to(\mu X.T\to T\to X)$ are equal as trees but inequivalent.
- So we switch to a coinductive view of equivalence.
- now equivalence coincides with equality of infinite tree expansions (prove it!)
- Now we need to *decide* this coinductive equivalence
  - show simulation algorithm (and its ML realization)
  - prove that it is sound, complete, and terminating

notice that we need to deal with more than just type equivalence: we also need to provide an `expose` function that unrolls recursive types as needed during typing. What needs to be proved about this?

Algorithmic rules for equi-recursive types $\to$ B μ

Algorithmic type equivalence $\boxed{\Gamma \mapsto S \leftrightarrow T}$

$$\frac{(S \leftrightarrow T) \in \mathcal{P}}{\mathcal{P} \mapsto S \leftrightarrow T} \quad \text{(QA-LOOP)}$$

$$\frac{\mathcal{P}, (\mu X.S_1 \leftrightarrow T) \mapsto \{X \mapsto \mu X.S_1\}S_1 \leftrightarrow T}{\mathcal{P} \mapsto \mu X.S_1 \leftrightarrow T} \quad \text{(QA-RECL)}$$

$$\frac{\mathcal{P}, (S \leftrightarrow \mu X.T_1) \mapsto S \leftrightarrow \{X \mapsto \mu X.T_1\}T_1}{\mathcal{P} \mapsto S \leftrightarrow \mu X.T_1} \quad \text{(QA-RECR)}$$

$$\mathcal{P} \mapsto B \leftrightarrow B \quad \text{(QA-BASE)}$$

$$\frac{\mathcal{P} \mapsto S_1 \leftrightarrow T_1 \qquad \mathcal{P} \mapsto S_2 \leftrightarrow T_2}{\mathcal{P} \mapsto S_1\to S_2 \leftrightarrow T_1\to T_2} \quad \text{(QA-ARROW)}$$

Exposure $\boxed{\Gamma \mapsto T \Uparrow T'}$

$$\frac{\mapsto \{X \mapsto \mu X.T_1\}T_1 \Uparrow T'}{\mapsto \mu X.T_1 \Uparrow T'} \quad \text{(XA-REC)}$$

$$\frac{T \text{ is not a recursive type}}{\mapsto T \Uparrow T} \quad \text{(XA-OTHER)}$$

Algorithmic typing $\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \quad \text{(TA-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathrel{|\!\!\blacktriangleright} \mathtt{T}_1 \Uparrow \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \mathrel{|\!\!\blacktriangleright} \mathtt{T}_2 \leftrightarrow \mathtt{T}_{11}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(TA-APP)}$$

ML Implementation

Datatypes

```
type ty =
    TyArr of ty * ty
  | TyId of string
  | TyVar of int * int
  | TyRec of string * ty

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

Type substitution

Type equivalence

```
let rec tyeqv pairs ctx tyS tyT =
  List.mem (tyS,tyT) pairs
  || match (tyS,tyT) with
    (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
       (tyeqv pairs ctx tyS1 tyT1) && (tyeqv pairs ctx tyS2 tyT2)
  | (TyId(b1),TyId(b2)) → b1=b2
  | (TyVar(i,_),TyVar(j,_)) → i=j
  | (TyRec(x,tyS1),_) →
       tyeqv ((tyS,tyT)::pairs) ctx (tysubstsnip tyS tyS1) tyT
  | (_,TyRec(x,tyT1)) →
       tyeqv ((tyS,tyT)::pairs) ctx tyS (tysubstsnip tyT tyT1)
  | _ → false

let tyeqv ctx tyS tyT =
  tyeqv [] ctx tyS tyT
```

Typing

```
let rec typeof ctx t =
  match t with
    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 (-1) tyT)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if tyeqv ctx tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
```

15.3 Iso-recursive Types

A different, and even simpler, way of dealing with recursive types is to "make the isomorphism explicit"...

λμ : Iso-recursive types → B μ

New syntactic forms

t	::=	...	*terms*
		fold [T] t	*folding*
		unfold [T] t	*unfolding*
v	::=	...	*values*
		fold [T] v	*folding*
T	::=	...	*types*
		X	*type variable*
		μX.T	*recursive type*
Γ	::=	...	*contexts*
		Γ,X	*type variable binding*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{unfold [S] (fold [T] }\texttt{v}_1\texttt{)} \longrightarrow \texttt{v}_1 \qquad \text{(E-FoldBeta)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{fold\ [T]\ t_1} \longrightarrow \mathtt{fold\ [T]\ t_1'}} \quad \text{(E-FOLD)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{unfold\ [T]\ t_1} \longrightarrow \mathtt{unfold\ [T]\ t_1'}} \quad \text{(E-UNFOLD)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t : T}}$

$$\frac{\mathtt{U} = \mu\mathtt{X.T_1} \qquad \Gamma \vdash \mathtt{t_1 : \{X \mapsto U\}T_1}}{\Gamma \vdash \mathtt{fold\ [U]\ t_1 : U}} \quad \text{(T-FOLD)}$$

$$\frac{\mathtt{U} = \mu\mathtt{X.T_1} \qquad \Gamma \vdash \mathtt{t_1 : U}}{\Gamma \vdash \mathtt{unfold\ [U]\ t_1 : \{X \mapsto U\}T_1}} \quad \text{(T-UNFOLD)}$$

15.3.1 Exercise: Re-do the examples in Section 15.1 using the Using the `fullisorec` checker from the course web directory. Solution on page 293. □

Prove: preservation and progress

Nice exercise: observe that we can just erase folds and unfolds from a well-typed program to obtain a well-typed program in the equi-recursive system. This tells us that any well typed iso-recursive program is “meaningful.” Does this constitute “soundness”? (Answer: no, it doesn’t tell us that the fold and unfold typing rules were done properly.)

15.4 Subtyping and Recursive Types

Chapter 16

Case Study: Featherweight Java

This chapter will draw some connections between the material that's been presented so far and the mainstream world of object-oriented type systems, focusing on Java. The pedagogical vehicle will be the Featherweight Java *language studied by Atsushi Igarashi, Phil Wadler, and myself [IPW99]. It is a very simple core calculus, not much larger than the lambda-calculus, omitting nearly all of the features of the full Java language (including assignment!), while retaining its basic flavor: classes, objects, methods, fields, (explicitly declared) subtyping, casts, etc. The presentation here will be based closely on (the full version of) the OOPSLA paper on FJ.*

The main technical work to be done lies in tying FJ as explicitly as possible with the concepts we have already seen. This is not all that simple, since we've been working in an entirely "structural" setting, where the only salient thing about a type expression is its structure (i.e., there are no "type names" except as simple abbreviations for structures), while Java uses names throughout its type system. A section re-formulating the simply typed lambda-calculus with subtyping in "by-name form" should provide a helpful bridge.

Sections:

- *By-name vs. structural presentations of type systems (note a big advantage of structural presentations: portability! If you're going to transmit code across the network, then what do the names mean??)*
- *A by-name presentation of the simply typed lambda-calculus with subtyping*

  (Comments about Pascal vs. M3 treatments of "branding".)
- *Summary of Featherweight Java*

Chapter 17

Type Reconstruction

For technical reasons, this chapter uses a slightly different definition of substitution from what we had before. This should be changed to correspond exactly to the earlier notion. Aside from that, it's essentially finished.

The present chapter does not mention let-polymorphism. I think that's a shame and I've tried to figure out how to put in something about it, but it's apparently quite hard to do it in a rigorous way without going on for page after page of technicalities.

Given an explicitly typed term in the simply typed lambda-calculus, we have seen an algorithm for determining its type, if it has one. In this chapter, we develop a more powerful **type reconstruction** algorithm, capable of calculating a **principal type** for a term in which some of the explicit type annotations are replaced by variables. Similar algorithms lie at the heart of languages like ML and Haskell.

The term "**type inference**" is often used instead of "type reconstruction."

17.1 Substitution

We will work in this chapter with the system $\lambda{\rightarrow}\mathbb{N}\mathtt{X}$, the simply typed lambda calculus with numbers and an infinite collection of atomic types. When we saw them in Section 9.1, these atomic types were completely uninterpreted. Now we are going to think of them as **type variables**—i.e., placeholders standing for other types. In order to make this idea precise, we need to define what it means to substitute arbitrary types for type variables.

17.1.1 Definition: A **type substitution** (or, for purposes of this chapter, just **substitution**) is a finite function from type variables to types. For example, we write $\{\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{U}\}$ for the substitution that maps `X` to `T` and `Y` to `U` and is undefined on other arguments.

We can regard a substitution σ as a function from types to types in an obvious way:

$$\begin{array}{lcl} \sigma\mathtt{X} & = & \begin{cases} \mathtt{T} & \text{if } \sigma \text{ maps } \mathtt{X} \text{ to } \mathtt{T} \\ \mathtt{X} & \text{if } \mathtt{X} \text{ is not in the domain of } \sigma \end{cases} \\ \sigma(\mathtt{Nat}) & = & \mathtt{Nat} \\ \sigma(\mathtt{T}_1 \to \mathtt{T}_2) & = & \sigma\mathtt{T}_1 \to \sigma\mathtt{T}_2 \end{array}$$

Note that we do not need to make any special provisions to avoid variable capture during type substitution, because there are no binders for type variables. (This will change when we discuss System F in Chapter 18; we will then have to treat type substitutions more carefully.)

Substitution is extended to contexts by defining $(\sigma\Gamma)(\mathtt{x})$ to be $\sigma(\Gamma(\mathtt{x}))$. Similarly, a type substitution is applied to a term `t` by applying it to all types appearing in `t`.

If σ and γ are type substitutions, we write $\sigma \circ \gamma$ for the type substitution formed by composing σ and γ as follows:

$$(\sigma \circ \gamma)(\mathtt{X}) = \begin{cases} \sigma(\gamma(\mathtt{X})) & \text{if } \mathtt{X} \in \mathit{dom}(\gamma) \\ \sigma(\mathtt{X}) & \text{if } \mathtt{X} \notin \mathit{dom}(\gamma) \text{ and } \mathtt{X} \in \mathit{dom}(\sigma) \\ \text{undefined} & \text{if } \mathtt{X} \notin \mathit{dom}(\gamma) \cup \mathit{dom}(\sigma). \end{cases}$$

Note that $(\sigma \circ \gamma)\mathtt{T} = \sigma(\gamma\mathtt{T})$. □

A crucial property of type substitutions is that they preserve the validity of typing statements: if a term involving variables is well typed, then so are all of its substitution instances.

17.1.2 Theorem [Preservation of typing under substitution]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and σ is any type substitution, then $\sigma\Gamma \vdash \sigma\mathtt{t} : \sigma\mathtt{T}$. □

Proof: Straightforward induction on typing derivations. □

17.2 Universal vs. Existential Type Variables

Suppose that `t` is a term containing type variables and Γ is an associated environment (possibly also containing type variables). There are two quite different questions that we can ask about `t`:

1. "Are *all* substitution instances of `t` well typed?" That is, is it the case that, for every σ, we have $\sigma\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$ for some `T`?

2. "Is *some* substitution instance of `t` well typed?" That is, can we find a σ such that $\sigma\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$ for some `T`?

According to the first view, type variables should be *held abstract* during typechecking, thus ensuring that a well-typed term will behave properly no matter what concrete types are later substituted for its type variables. For example, the term

```
λf:X→X. λa:X. f (f a);
```

has type `(X→X)→X→X`, and, whenever we replace `X` by `T`, the instance

```
λf:T→T. λa:T. f (f a);
```

is well typed. Holding type variables abstract in this way leads to the idea of **parametric polymorphism**, in which type variables are used to encode the fact that a term can be used in many concrete contexts with different concrete types. We shall return to parametric polymorphism in more depth in Chapter 18.

In the second view, the original term `t` may not even be well typed; what we want to know is whether it can be *instantiated* to a well typed term by choosing appropriate values for some of its type variables. For example, the term

```
λf:Y. λa:X. f (f a);
```

is not typeable as it stands, but if we replace `Y` by `Nat→Nat` and `X` by `Nat`, we obtain

```
λf:Nat→Nat. λa:Nat. f (f a);
```

of type `(Nat→Nat)→Nat→Nat`. Or, if we simply replace `Y` by `X→X`, we obtain the term

```
λf:X→X. λa:X. f (f a);
```

which is well typed even though it contains variables. Indeed, this term is a **most general** instance of `λf:Y. λa:X. f (f a)`, in the sense that it makes the least commitment about the values of type variables that is required to obtain a well-typed term.

We may even, in the limit, allow type annotations to be omitted completely, filling in a fresh type variable during parsing whenever an annotation is discovered to be missing. This allows the term above to be written

```
λf. λa. f (f a);
```

as in languages like ML.

Looking for valid instantiations of type variables leads to the idea of **type reconstruction**, in which the compiler is asked to help fill in type information that has been underspecified by the programmer. We develop this point of view in the rest of the chapter.

17.2.1 Definition: Let Γ be a context and $\mathtt{t}$ a term. A **typing** for $(\Gamma, \mathtt{t})$ is a pair $(\sigma, \mathtt{T})$ such that $\sigma\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$. □

17.2.2 Example: Let $\Gamma = \mathtt{f}{:}\mathtt{X}, \mathtt{a}{:}\mathtt{Y}$ and $\mathtt{t} = \mathtt{f\ a}$. Then

$(\{\mathtt{X} \mapsto \mathtt{Y}{\to}\mathtt{Nat}\}, \mathtt{Nat})$
$(\{\mathtt{X} \mapsto \mathtt{Y}{\to}\mathtt{Z}, \mathtt{Z} \mapsto \mathtt{Nat}\}, \mathtt{Z})$
$(\{\mathtt{X} \mapsto \mathtt{Y}{\to}\mathtt{Nat}{\to}\mathtt{Nat}\}, \mathtt{Nat}{\to}\mathtt{Nat})$
$(\{\mathtt{X} \mapsto \mathtt{Y}{\to}\mathtt{Z}\}, \mathtt{Z})$

are all typings for $(\Gamma, \mathtt{t})$. □

17.2.3 Exercise [Quick check]: Find three different typings for

$$\lambda\mathtt{x{:}X.}\ \lambda\mathtt{y{:}Y.}\ \lambda\mathtt{z{:}Z.}\ \mathtt{(x\ z)\ (y\ z).}$$

with respect to the empty context. □

17.3 Constraint-Based Typing

We now give a different presentation of the typing relation in which, for example, instead of checking directly whether types of arguments match domains of functions, we generate a set of constraints C recording the fact that this check should be performed later.

17.3.1 Definition: A **constraint set** C is a set of equations $\{\mathtt{S}_i = \mathtt{T}_i\ ^{i\in 1..n}\}$. A substitution σ is said to **satisfy** the constraint set C if, for each i, the substitution instances $\sigma\mathtt{S}_i$ and $\sigma\mathtt{T}_i$ are equal. □

17.3.2 Definition: The **constraint typing relation** $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$ is defined by the rules below. Informally, $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$ can be read as "Term t has type T under assumptions Γ whenever constraints C are satisfied." The subscript $\mathcal{X}$, which tracks the type variables introduced in the process, is used for internal bookkeeping—to make sure that the fresh type variables used in different subderivations are actually distinct.

$$\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \mid_{\emptyset} \{\}} \qquad \text{(CT-VAR)}$$

$$\frac{\Gamma, \mathtt{x{:}S} \vdash \mathtt{t}_1 : \mathtt{T} \mid_{\mathcal{X}} C \qquad \mathtt{x} \notin dom(\Gamma)}{\Gamma \vdash \lambda\mathtt{x{:}S.t}_1 : \mathtt{S}\to\mathtt{T} \mid_{\mathcal{X}} C} \qquad \text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset \qquad \mathtt{X} \text{ not mentioned in } \mathcal{X}_1, \mathcal{X}_2, \mathtt{T}_1, \mathtt{T}_2, C_1, C_2, \Gamma, \mathtt{t}_1, \text{or } \mathtt{t}_2\end{array}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{X} \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathtt{X}\}} C_1 \cup C_2 \cup \{\mathtt{T}_1 = \mathtt{T}_2\to\mathtt{X}\}} \qquad \text{(CT-APP)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \mid_{\mathcal{X}} C}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat} \mid_{\mathcal{X}} C \cup \{\mathtt{T} = \mathtt{Nat}\}} \qquad \text{(CT-SUCC)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}_2} C_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \mid_{\mathcal{X}_3} C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\mathtt{T}_1 = \mathtt{Nat}, \mathtt{T}_2 = \mathtt{T}_3\to\mathtt{T}_3\}\end{array}}{\Gamma \vdash \mathtt{iter}\ \mathtt{T}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3 : \mathtt{T}_3 \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3} C'} \qquad \text{(CT-ITER)}$$

As usual, these rules (when read from bottom to top) determine a straightforward procedure that, given Γ and t, calculates T and C (and $\mathcal{X}$) such that $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$.

However, unlike the original typing algorithm, this one never fails, in the sense that for every Γ and $\mathtt{t}$ there are always some $\mathtt{T}$ and C such that $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid C$, and moreover that $\mathtt{T}$ and C are uniquely determined by Γ and $\mathtt{t}$, up to the choice of names of fresh type variables in $\mathcal{X}$. (The nondeterminism arising from the freedom to choose different fresh type variable names will be addressed in Exercise 17.3.9.)

To lighten the notation in the following discussion, we usually elide the $\mathcal{X}$ and write just $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid C$. □

17.3.3 Exercise [Quick check]: Construct a constraint typing derivation whose conclusion is

$$\vdash \lambda\mathtt{x}{:}\mathtt{X}.\ \lambda\mathtt{y}{:}\mathtt{Y}.\ \lambda\mathtt{z}{:}\mathtt{Z}.\ \mathtt{(x\ z)\ (y\ z)} : \mathtt{S} \mid C$$

for some $\mathtt{S}$ and C. □

17.3.4 Definition: Suppose that $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$. A **typing** for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ is a pair $(\sigma, \mathtt{T})$ such that σ satisfies C and $\sigma\mathtt{S} = \mathtt{T}$. □

Given a context Γ and a term $\mathtt{t}$, we now have two different ways of calculating sets of typings for Γ and $\mathtt{t}$:

1. directly, via Definition 17.2.1; or
2. via the constraint typing relation, by finding $\mathtt{S}$ and C such that $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$ and then using Definition 17.3.4 to identify the set of typings for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$.

Our next job is to show that these two methods yield essentially the same sets of typings. We do this in two steps. First we show that every $(\Gamma, \mathtt{t}, \mathtt{S}, C)$-typing is a $(\Gamma, \mathtt{t})$-typing [soundness]. Then we show that every $(\Gamma, \mathtt{t})$-typing can be extended to a $(\Gamma, \mathtt{t}, \mathtt{S}, C)$-typing [completeness].

17.3.5 Theorem [Soundness of constraint typing]: Suppose that $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$. If $(\sigma, \mathtt{T})$ is a typing for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$, then it is also a typing for $(\Gamma, \mathtt{t})$. □

Proof: By induction on the given constraint typing derivation for $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$, reasoning by cases on the last rule used.

Case CT-VAR: $\mathtt{t} = \mathtt{x}$
$\mathtt{x}{:}\mathtt{S} \in \Gamma$
$C = \{\}$

The result is immediate, since by T-VAR $\sigma\Gamma \vdash \mathtt{x} : (\sigma\Gamma)(\mathtt{x})$ for any σ.

Case CT-ABS: $\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{T_1}.\mathtt{t_1}$
$\mathtt{S} = \mathtt{T_1}{\to}\mathtt{S_2}$
$\Gamma, \mathtt{x}{:}\mathtt{T_1} \vdash \mathtt{t_1} : \mathtt{S_2} \mid C$

By the induction hypothesis, $(\sigma, \sigma\mathtt{S_2})$ is a typing for $((\Gamma, \mathtt{x}{:}\mathtt{T_1}), \mathtt{t_1})$, i.e.,

$$\sigma\Gamma, \mathtt{x}{:}\sigma\mathtt{T_1} \vdash \sigma\mathtt{t_1} : \sigma\mathtt{S_2}.$$

By T-ABS, $\sigma\Gamma \vdash \lambda\mathtt{x}{:}\sigma\mathtt{T_1}.\sigma\mathtt{t_1} : \sigma\mathtt{T_1}{\to}\sigma\mathtt{S_2} = \sigma(\mathtt{T_1}{\to}\mathtt{S_2}) = \mathtt{T}$, as required.

Case CT-APP:
$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\mathtt{S} = \mathtt{X}$
$\Gamma \vdash \mathtt{t}_1 : \mathtt{S}_1 \mid \mathrm{C}_1$
$\Gamma \vdash \mathtt{t}_2 : \mathtt{S}_2 \mid \mathrm{C}_2$
$\mathrm{C} = \mathrm{C}_1 \cup \mathrm{C}_2 \cup \{\mathtt{S}_1 = \mathtt{S}_2 \to \mathtt{X}\}$

By the definition of satisfaction, σ satisfies both C_1 and C_2 and $\sigma\mathtt{S}_1 = \sigma(\mathtt{S}_2\to\mathtt{X})$. By Definition 17.3.4, we see that $(\sigma, \sigma\mathtt{S}_1)$ and $(\sigma, \sigma\mathtt{S}_2)$ are typings for $(\Gamma, \mathtt{t}_1, \mathtt{S}_1, \mathrm{C}_1)$ and $(\Gamma, \mathtt{t}_2, \mathtt{S}_2, \mathrm{C}_2)$, from which the induction hypothesis gives us $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathtt{S}_1$ and $\sigma\Gamma \vdash \sigma\mathtt{t}_2 : \sigma\mathtt{S}_2$. But since $\sigma\mathtt{S}_1 = \sigma\mathtt{S}_2\to\sigma\mathtt{X}$, we then have $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathtt{S}_2\to\sigma\mathtt{X}$, and, by T-APP, $\sigma\Gamma \vdash \sigma(\mathtt{t}_1\ \mathtt{t}_2) : \sigma\mathtt{X} = \mathtt{T}$, as required.

Other cases:

Left as an exercise. □

17.3.6 Definition: Write $\sigma\backslash\mathcal{X}$ for the substitution that is undefined for all the variables in $\mathcal{X}$ and otherwise behaves like σ. □

17.3.7 Theorem [Completeness of constraint typing]: Suppose $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid_{\mathcal{X}} \mathrm{C}$. If $(\sigma, \mathtt{T})$ is a typing for $(\Gamma, \mathtt{t})$ and $dom(\sigma) \cap \mathcal{X} = \emptyset$, then there is some typing $(\sigma', \mathtt{T})$ for $(\Gamma, \mathtt{t}, \mathtt{S}, \mathrm{C})$ such that $\sigma'\backslash\mathcal{X} = \sigma$. □

Proof: By induction on the given constraint typing derivation.

Case CT-VAR:
$\mathtt{t} = \mathtt{x}$
$\mathtt{x}{:}\mathtt{S} \in \Gamma$

From the assumption that $(\sigma, \mathtt{T})$ is a typing for $(\Gamma, \mathtt{x})$, the inversion lemma for the typing relation (7.4.1) tells us that $\mathtt{T}$ is $(\sigma\Gamma)(\mathtt{x})$. But then $(\sigma, \mathtt{T})$ is also a $(\Gamma, \mathtt{x}, \mathtt{S}, \{\})$-typing.

Case CT-ABS:
$\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1$
$\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_1 : \mathtt{S}_2 \mid_{\mathcal{X}} \mathrm{C}$
$\mathtt{S} = \mathtt{T}_1\to\mathtt{S}_2$

From the assumption that $(\sigma, \mathtt{T})$ is a typing for $(\Gamma, \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1)$, the inversion lemma for the typing relation yields $\sigma\Gamma, \mathtt{x}{:}\sigma\mathtt{T}_1 \vdash \sigma\mathtt{t}_1 : \mathtt{T}_2$ and $\mathtt{T} = \sigma\mathtt{T}_1\to\mathtt{T}_2$.

By the induction hypothesis, there is a typing $(\sigma', \mathtt{T}_2)$ for $((\Gamma, \mathtt{x}{:}\mathtt{T}_1), \mathtt{t}_1, \mathtt{S}_2, \mathrm{C})$ such that $\sigma'\backslash\mathcal{X}$ agrees with σ. But then $\sigma'(\mathtt{S}) = \sigma'(\mathtt{T}_1\to\mathtt{S}_2) = \sigma\mathtt{T}_1\to\sigma'\mathtt{S}_2 = \sigma\mathtt{T}_1\to\mathtt{T}_2 = \mathtt{T}$, so $(\sigma', \mathtt{T})$ is a typing for $(\Gamma, (\lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1), \mathtt{T}_1\to\mathtt{S}_2, \mathrm{C})$.

Case CT-APP:
$\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\Gamma \vdash \mathtt{t}_1 : \mathtt{S}_1 \mid_{\mathcal{X}_1} \mathrm{C}_1$
$\Gamma \vdash \mathtt{t}_2 : \mathtt{S}_2 \mid_{\mathcal{X}_2} \mathrm{C}_2$
$\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$ and $\mathtt{X}$ not mentioned in $\mathcal{X}_1, \mathcal{X}_2, \mathtt{S}_1, \mathtt{S}_2, \mathrm{C}_1, \mathrm{C}_2$
$\mathtt{S} = \mathtt{X}$
$\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathtt{X}\}$
$\mathrm{C} = \mathrm{C}_1 \cup \mathrm{C}_2 \cup \{\mathtt{S}_1 = \mathtt{S}_2\to\mathtt{X}\}$

From the assumption that $(\sigma, \mathtt{T})$ is a typing for $(\Gamma, \mathtt{t}_1\ \mathtt{t}_2)$, the inversion lemma for the typing relation yields $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}$ and $\sigma\Gamma \vdash \sigma\mathtt{t}_2 : \mathtt{T}_1$.

By the induction hypothesis, there are typings $(\sigma_1, \mathtt{T}_1 \to \mathtt{T})$ for $(\Gamma, \mathtt{t}_1, \mathtt{S}_1, C_1)$ and $(\sigma_2, \mathtt{T}_1)$ for $(\Gamma, \mathtt{t}_2, \mathtt{S}_2, C_2)$, and $\sigma_1 \backslash \mathcal{X}$ and $\sigma_2 \backslash \mathcal{X}$ agree with σ. We must exhibit a substitution σ' such that: (1) $\sigma' \backslash \mathcal{X}$ agrees with σ, (2) $\sigma'\mathtt{X} = \mathtt{T}$, and (3) σ' satisfies $\{\mathtt{S}_1 = \mathtt{S}_2 \to \mathtt{X}\}$, i.e., $\sigma'\mathtt{S}_1 = \sigma'\mathtt{S}_2 \to \sigma'\mathtt{X}$. Define σ' as follows:

$$\begin{array}{lcll} \sigma'\mathtt{Y} & = & \sigma\mathtt{Y} & \text{if } \mathtt{Y} \notin \mathcal{X} \\ \sigma'\mathtt{Y} & = & \sigma_1\mathtt{Y} & \text{if } \mathtt{Y} \in \mathcal{X}_1 \\ \sigma'\mathtt{Y} & = & \sigma_2\mathtt{Y} & \text{if } \mathtt{Y} \in \mathcal{X}_2 \\ \sigma'\mathtt{Y} & = & \mathtt{T} & \text{if } \mathtt{Y} = \mathtt{X}. \end{array}$$

Conditions (1) and (2) are obviously satisfied. To check (3), calculate as follows: $\sigma'\mathtt{S}_1 = \sigma_1\mathtt{S}_1 = \mathtt{T}_1 \to \mathtt{T} = \sigma_2\mathtt{S}_2 \to \mathtt{T} = \sigma'\mathtt{S}_2 \to \sigma'\mathtt{X} = \sigma'(\mathtt{S}_2 \to \mathtt{X})$.

Other cases:

Left as an exercise. □

17.3.8 Corollary: Suppose $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid C$. There is some typing for $(\Gamma, \mathtt{t})$ iff there is some typing for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$. □

Proof: By Theorems 17.3.5 and 17.3.7. □

17.3.9 Exercise [Recommended]: Because we have not described how the choice of fresh variable names actually occurs (beyond stipulating that they must be "fresh enough"), the constraint generation algorithm that we have been working with is actually nondeterministic. Fortunately, this nondeterminism is inessential and easily eliminated.

In a production compiler, the nondeterministic choice of a fresh type variable name in the rule CT-APP might be replaced by a call to a function that generates a new type variable—different from all others that it ever generates, and from all type variables mentioned explicitly in the context or term being checked—each time it is called. Because this global "`gensym`" operation works by side effects on a hidden global variable, it is difficult to reason about it formally. However, we can easily simulate it by "threading" a sequence of unused variable names through the constraint generation rules.

Let $\mathtt{F}$ denote a sequence of fresh type variable names (with each element of the sequence different from every other). Then, instead of writing

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C$$

for the constraint generation judgement, we write

$$\Gamma \vdash_{\mathtt{F}} \mathtt{t} : \mathtt{T} \mid_{\mathtt{F}'} C,$$

where Γ, $\mathtt{F}$, and $\mathtt{t}$ are inputs to the algorithm and $\mathtt{T}$, $\mathtt{F}'$, and C are outputs. Whenever it needs a fresh type variable, the algorithm takes off the front element of $\mathtt{F}$ and returns the rest of $\mathtt{F}$ as $\mathtt{F}'$.

Write out the rules for this algorithm in detail. Prove that they are equivalent, in an appropriate sense, to the original constraint generation rules. Solution on page 289. □

17.3.10 Exercise [Recommended]: Implement the algorithm discussed in Exercise 17.3.9 in ML. Use the datatype

```
type ty =
    TyArr of ty * ty
  | TyId of string
  | TyBool
  | TyNat
```

for types, and

```
type constr = (ty * ty) list
```

for constraint sets. You will also need a representation for infinite sequences of fresh variable names. There are lots of ways of doing this; here is a fairly direct one using a recursive datatype:

```
type nextuvar = NextUVar of string * uvargenerator
and uvargenerator = unit → nextuvar

let uvargen =
  let rec f n () = NextUVar("?X_" ^ string_of_int n, f (n+1))
  in f 0
```

That is, `uvargen` is a function that, when called with argument `()`, returns a value of the form `NextUVar(x,f)`, where `x` is a fresh type variable name and `f` is another function of the same form. Solution on page 290. □

17.4 Unification

We have given two different characterizations of the set of typings for a given term in a given context. But we have not addressed the algorithmic problem of deciding whether or not this set is empty—i.e., of deciding whether it is possible to replace type variables by types so as to make the term typeable in the ordinary sense. The key insight that we need, due to Hindley [?] and Milner [?], is that, if the set of typings of a term is nonempty, it always contains a "best" element, in the sense that the rest of the typings can be generated straightforwardly from this one.

To show that such **principal typings** exist (and, in fact, to give an algorithm for generating them), we rely on the familiar operation of **unification** [?] on constraint sets.

17.4.1 Definition: A substitution σ is said to **unify** two types $\mathtt{S}$ and $\mathtt{T}$ if $\sigma\mathtt{S} = \sigma\mathtt{T}$. So saying that "$\sigma$ satisfies the constraint set C" is the same as saying "σ unifies $\mathtt{S}_i$ and $\mathtt{T}_i$ for every equation $\mathtt{S}_i = \mathtt{T}_i$ in C." We sometimes speak of σ as a **unifier** for C. □

17.4.2 Definition: We say that a substitution σ is **more general than** a substitution σ', written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some substitution γ. □

17.4.3 Definition: A **principal unifier** for a constraint set C is a substitution σ that satisfies C and such that $\sigma \sqsubseteq \sigma'$ for every substitution σ' that also satisfies C. □

17.4.4 Exercise [Quick check]: Write down principal unifiers (when they exist) for the following sets of constraints:

$\{\mathtt{X} = \mathtt{Nat},\ \mathtt{Y} = \mathtt{X}{\to}\mathtt{X}\}$
$\{\mathtt{Nat}{\to}\mathtt{Nat} = \mathtt{X}{\to}\mathtt{Y}\}$
$\{\mathtt{X}{\to}\mathtt{Y} = \mathtt{Y}{\to}\mathtt{Z},\ \mathtt{Z} = \mathtt{U}{\to}\mathtt{W}\}$
$\{\mathtt{Nat} = \mathtt{Nat}{\to}\mathtt{Y}\}$
$\{\mathtt{Y} = \mathtt{Nat}{\to}\mathtt{Y}\}$
$\{\}$ (the empty set of constraints)

Solution on page 291. □

17.4.5 Definition: The **unification algorithm** is defined as follows:

$$
\begin{array}{lll}
\textit{unify}(C) & = & \text{if } C = \emptyset, \text{ then } \{\,\} \\
 & & \text{else let } \{\mathtt{S} = \mathtt{T}\} \cup C' = C \text{ in} \\
 & & \quad \text{if } \mathtt{S} = \mathtt{T} \\
 & & \qquad \text{then } \textit{unify}(C') \\
 & & \quad \text{else if } \mathtt{S} = \mathtt{X} \text{ and } \mathtt{X} \notin \mathit{FV}(\mathtt{T}) \\
 & & \qquad \text{then } \textit{unify}(\{\mathtt{X} \mapsto \mathtt{T}\}C') \circ \{\mathtt{X} \mapsto \mathtt{T}\} \\
 & & \quad \text{else if } \mathtt{T} = \mathtt{X} \text{ and } \mathtt{X} \notin \mathit{FV}(\mathtt{S}) \\
 & & \qquad \text{then } \textit{unify}(\{\mathtt{X} \mapsto \mathtt{S}\}C') \circ \{\mathtt{X} \mapsto \mathtt{S}\} \\
 & & \quad \text{else if } \mathtt{S} = \mathtt{S}_1{\to}\mathtt{S}_2 \text{ and } \mathtt{T} = \mathtt{T}_1{\to}\mathtt{T}_2 \\
 & & \qquad \text{then } \textit{unify}(C' \cup \{\mathtt{S}_1 = \mathtt{T}_1, \mathtt{S}_2 = \mathtt{T}_2\}) \\
 & & \quad \text{else} \\
 & & \qquad \text{fail}
\end{array}
$$

where σC is the constraint set formed by applying σ to both sides of all the constraints in C. □

17.4.6 Theorem: The algorithm *unify* always terminates, fails only when given a non-unifiable constraint set as input, and otherwise returns a principal unifier. More formally:

1. *unify*(C) halts, either by failing or by returning a substitution, for all C;

2. if $unify(C) = \sigma$, then σ is a unifier for C;

3. if δ is a unifier for C, then $unify(C) = \sigma$ with $\sigma \sqsubseteq \delta$. □

Proof: For part (1), define the *degree* of a constraint set C to be the pair (m, n), where m is the number of distinct type variables in C and n is the total size of the types in C. It is easy to check that each clause of the *unify* algorithm either terminates immediately (with success in the first case or failure in the last) or else makes a recursive call to *unify* with a constraint set of smaller degree.

Part (2) is a straightforward induction on the number of recursive calls in the computation of $unify(C)$. All the cases are trivial except for the two involving variables, which depend on the observation that, if σ unifies $\{X \mapsto T\}C'$, then $\sigma \circ \{X \mapsto T\}$ unifies $\{X = T\} \cup C'$.

Part (3) again proceeds by induction on the number of recursive calls in the computation of $unify(C)$. If C is empty, then $unify(C)$ immediately returns $\{\}$; since $\delta = \delta \circ \{\}$, we have $\{\} \sqsubseteq \delta$ as required. If C is non-empty, then $unify(C)$ chooses some pair (S, T) from C and continues by cases on the shapes of S and T.

Case: $S = T$

Since δ is a unifier for C, it also unifies C'. By the induction hypothesis, $unify(C) = \sigma$ with $\sigma \sqsubseteq \delta$, as required.

Case: $S = X$ and $X \notin T$

Since δ unifies S and T, we have $\delta(X) = \delta(T)$. So, for any type U, we have $\delta(U) = \delta(\{X \mapsto T\}U)$; in particular, since δ satisfies C' it must also satisfy $\{X \mapsto T\}C'$. The induction hypothesis then tells us that $unify(\{X \mapsto T\}C') = \sigma'$, with $\delta = \gamma \circ \sigma'$ for some γ. Since $unify(C) = \sigma' \circ \{X \mapsto T\}$, showing that $\delta = \gamma \circ (\sigma' \circ \{X \mapsto T\})$ will complete the argument. Choose any type variable Y. If $Y \neq X$, then clearly $(\gamma \circ (\sigma' \circ \{X \mapsto T\}))Y = (\gamma \circ \sigma')Y = \delta Y$. On the other hand, $(\gamma \circ (\sigma' \circ \{X \mapsto T\}))X = (\gamma \circ \sigma')T = \delta X$, as we saw above. Combining these observations, we see that $\delta Y = (\gamma \circ (\sigma' \circ \{X \mapsto T\}))Y$ for all variables Y, i.e. $\delta = (\gamma \circ (\sigma' \circ \{X \mapsto T\}))$.

Case: $T = X$ and $X \notin S$

Similar.

Case: $S = S_1 \to S_2$ and $T = T_1 \to T_2$

Straightforward. Just note that δ is a unifier of $\{S_1 \to S_2 = T_1 \to T_2\} \cup C'$ iff it is a unifier of $C' \cup \{S_1 = T_1, S_2 = T_2\}$.

If none of the above cases apply to S and T, then $unify(C)$ fails. But this can only happen in two ways: either S is Nat and T is an arrow type (or vice versa), or else $S = X$ and $X \in T$ (or vice versa). The first case obviously contradicts the assumption that C is unifiable. To see that the second does too, recall that, by assumption, $\delta S = \delta T$; if X occurred in T, then δT would always be strictly larger than δS. Thus, $unify(C)$ never fails when C is satisfiable. □

17.4.7 Exercise [Recommended]: Implement the unification algorithm in ML.

The main data structure needed for this exercise is a representation of substitutions. There are many alternatives; one simple one is to reuse the `constr` datatype from Exercise 17.3.10: a substitution is just a constraint set, all of whose left-hand sides are unification variables. If `substinty` is a function that performs substitution of a type for a single type variable

```
let substinty tyX tyT tyS =
  let rec o = function
      TyArr(tyT1,tyT2) → TyArr(o tyT1, o tyT2)
    | TyNat → TyNat
    | TyBool → TyBool
    | TyId(s) → if s=tyX then tyT else TyId(s)
  in o tyS
```

then application of a whole substitution to a type can be defined as follows:

```
let applysubst constr tyT =
  List.fold_left
    (fun tyS (TyId(tyX),tyC2) → substinty tyX tyC2 tyS)
    tyT constr
```

Solution on page 291. □

17.5 Principal Typings

17.5.1 Definition: A **principal typing** for $(\Gamma, \mathtt{t}, \mathtt{S}, \mathtt{C})$ is a typing $(\sigma, \mathtt{T})$ such that, whenever $(\sigma', \mathtt{T}')$ is also a typing for $(\Gamma, \mathtt{t}, \mathtt{S}, \mathtt{C})$, we have $\sigma \sqsubseteq \sigma'$. □

17.5.2 Exercise [Quick check]: Find a principal typing for

```
λx:X. λy:Y. λz:Z. (x z) (y z).
```
□

17.5.3 Theorem [Principal typing]: If $(\Gamma, \mathtt{t}, \mathtt{S}, \mathtt{C})$ has any typing, then it has a principal one. Moreover, the unification algorithm can be used to determine whether $(\Gamma, \mathtt{t}, \mathtt{S}, \mathtt{C})$ has a typing and, if so, to return a principal one. □

Proof: Immediate from the definition of typing and the properties of unification. □

17.5.4 Corollary: It is decidable whether $(\Gamma, \mathtt{t})$ has a typing. □

Proof: By Corollary 17.3.8 and Theorem 17.5.3. □

17.5.5 Exercise [Recommended]: Use the implementations of constraint generation (Exercise 17.3.10) and unification (Exercise 17.4.7) to construct a running typechecker that calculates principal typings, using the `simple` checker provided in the course directory as a starting point. □

17.5.6 Exercise: What difficulties arise in extending the basic definitions (17.3.2, etc.) to deal with records? How might they be addressed? Solution on page 292. □

Principal typings can be used to build a type reconstruction algorithm that works more incrementally than the one we have developed here. Instead of generating all the constraints first and then trying to solve them, we can interleave generation and solving, so that the type reconstruction algorithm actually returns a principal typing at each step. The fact that the typings are always *principal* is what ensures that the algorithm never needs to go back and re-analyze a subterm that it has already processed, since it makes only the minimum commitments needed to achieve typeability at each step. One major advantage of such an algorithm is that it can pinpoint errors in the user's program much more precisely.

17.5.7 Exercise: Modify your solution to Exercise 17.5.5 to perform unification incrementally during typechecking and return principal typings. □

17.6 Further Reading

Chapter 18

Universal Types

Type structure is a syntactic discipline for enforcing levels of abstraction.
— John Reynolds [Rey83]

Some writing needed.

18.1 Motivation

As we observed at the beginning of Chapter 12, 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 12, 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 $\lambda^{\rightarrow}$ 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 26), it is sometimes called the **second-order lambda-calculus**.

18.2 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 12 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.

18.3 Definitions

The definition of the polymorphic lambda-calculus (System F 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 $\{\texttt{X} \mapsto \texttt{Nat}\}(\lambda\texttt{x:X.x})$, i.e., $\lambda\texttt{x:Nat.x}$, 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 $\mathtt{T}\to\mathtt{T}$; 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:

$$\mathtt{id} : \forall\mathtt{X}.\mathtt{X}\to\mathtt{X}$$

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

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{X}.\mathtt{t}_2 : \forall\mathtt{X}.\mathtt{T}_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall\mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}} \qquad \text{(T-TAPP)}$$

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.

System F : Polymorphic lambda-calculus $\to \forall$

Syntax

`t ::=`		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`λX.t`	*type abstraction*
	`t [T]`	*type application*
`v ::=`		*values*
	`λx:T.t`	*abstraction value*
	`λX.t`	*type abstraction value*
`T ::=`		*types*
	`X`	*type variable*
	`T→T`	*type of functions*
	`∀X.T`	*universal type*
`Γ ::=`		*contexts*

$\emptyset$	*empty context*
$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*
$\Gamma, \mathtt{X}$	*type variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_{12} \tag{E-Beta}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \tag{E-App1}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \tag{E-App2}$$

$$(\lambda \mathtt{X}.\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{t}_{12} \tag{E-Beta2}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1'\ [\mathtt{T}_2]} \tag{E-TApp}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \tag{T-Var}$$

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

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

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2} \tag{T-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}} \tag{T-TApp}$$

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.

18.3.1 Exercise [Quick check]: In Exercise 7.3.1, 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 System F? □

18.4 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 System F.
- 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 4 for the untyped lambda-calculus. Although these encodings are of little practical importance, they make excellent exercises for understanding the intricacies of System F.

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 7.4.5). In System F, 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
```

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

Polymorphic Lists

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)
```

18.4.2 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
```

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

18.4.4 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.

18.4.5 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. □

18.4.6 Exercise [Moderately difficult, recommended]: Use the functions defined in Exercise 18.4.5 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 $(n, n - 1)$...) □

18.4.7 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 System F 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.] □

18.5 Metatheory

Soundness

The preservation and progress properties from the simply typed lambda-calculus holds exactly the same here. To prove it, we need one additional substitution lemma, for types.

Strong Normalization

The strong normalization property for System F, proved using an extension of the method presented in Chapter 8, 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 [**?**].

18.5.1 Theorem [Strong normalization]: All well-typed System F terms are strongly normalizing. □

Erasure and Typeability

There are two reasonable definitions of erasure for System F: the "type-passing" and "type-forgetting" interpretations...

18.5.2 Exercise: The strong normalization property of System System F implies that the term

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

in the untyped lambda-calculus cannot be typed in System 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 F term **exposed** if it is a variable, an abstraction $\lambda\texttt{x:T.t}$, or an application $\texttt{t s}$ (i.e., if it is *not* a type abstraction $\lambda\texttt{X.t}$ or type application $\texttt{t S}$).

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

2. Write $\lambda\overline{\texttt{X}}\texttt{.t}$ as shorthand for a nested sequence of type abstractions of the form $\lambda\texttt{X}_1\ldots.\lambda\texttt{X}_n\texttt{.t}$. Similarly, write $\texttt{t }[\overline{\texttt{A}}]$ for a nested sequence of type applications $\texttt{((t [A}_1\texttt{])}\ldots[\texttt{A}_{n-1}]\texttt{) [A}_n\texttt{]}$ and $\forall\overline{\texttt{X}}\texttt{.T}$ for a nested sequence of polymorphic types $\forall\texttt{X}_1\ldots.\forall\texttt{X}_n\texttt{.T}$. Note that these sequences are allowed to be empty. For example, if $\overline{\texttt{X}}$ is the empty sequence of type variables, then $\forall\overline{\texttt{X}}\texttt{.T}$ is just $\texttt{T}$.

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

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

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

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

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

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

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

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

   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 $\textit{leftmost-leaf}(\mathtt{T}_1) = \mathtt{X}_i$ for some $\mathtt{X}_i \in \overline{\mathtt{X}}_1\overline{\mathtt{X}}_2$.

7. Show that Ω is not typeable in System F.

Solution on page **??**. □

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

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

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

Type Reconstruction

18.6 Implementation

Nameless Representation of Types

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 d t = tmshifti d 0 t

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

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

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

let tmsubstsnip s t = tmshift (-1) (tmsubst 0 (tmshift 1 s) t)

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

let tysubst tyS tyT = tysubsti tyS 0 tyT

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

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 (-1) (tytmsubst (tyshift 1 tyS) t)
```

18.7 Further Reading

Chapter 19

Existential Types

Some writing needed.

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.

19.1 Motivation

The polymorphic types in Chapter 18 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 System F in Chapter 18, 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 `{∃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:

[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:

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

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

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:

$$\frac{\Gamma \vdash \mathtt{t}_2 : \{\mathtt{X} \mapsto \mathtt{U}\}\mathtt{T}_2}{\Gamma \vdash \{\exists\mathtt{X}\text{=}\mathtt{U},\mathtt{t}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}_2\} : \{\exists\mathtt{X},\mathtt{T}_2\}} \qquad \text{(T-PACK)}$$

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}}
```

19.1.1 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:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \exists\mathtt{X}.\mathtt{T}_{12} \qquad \Gamma, \mathtt{X}, \mathtt{x}\text{:}\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\text{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \qquad \text{(T-UNPACK)}$$

That is, if $\mathtt{t}_1$ 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 $\mathtt{t}_2$.

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 5.1, 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 $\mathtt{t}_2$:

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.

Existential types $\rightarrow\forall\exists$

New syntactic forms

`t`	::=	...	*terms*
		$\{\exists\mathtt{X=T,t}\}$ `as T`	*packing*
		`let {X,x}=t in t`	*unpacking*
`v`	::=	...	*values*
		$\{\exists\mathtt{X=T,v}\}$ `as T`	*package value*
`T`	::=	...	*types*
		$\{\exists\mathtt{X,T}\}$	*existential type*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{let}\ \{\mathtt{X,x}\}=\{\exists\mathtt{X=T_{11},v_{12}}\}\ \mathtt{as}\ \mathtt{T_1}\ \mathtt{in}\ \mathtt{t_2} \longrightarrow \{\mathtt{X} \mapsto \mathtt{T_{11}}\}\{\mathtt{x} \mapsto \mathtt{v_{12}}\}\mathtt{t_{12}} \qquad \text{(E-PACKBETA)}$$

$$\frac{\mathtt{t_{12}} \longrightarrow \mathtt{t}'_{12}}{\{\exists\mathtt{X=T_{11},t_{12}}\}\ \mathtt{as}\ \mathtt{T_1} \longrightarrow \{\exists\mathtt{X=T_{11},t'_{12}}\}\ \mathtt{as}\ \mathtt{T_1}} \qquad \text{(E-PACK)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t}'_1}{\mathtt{let}\ \{\mathtt{X,x}\}=\mathtt{t_1}\ \mathtt{in}\ \mathtt{t_2} \longrightarrow \mathtt{let}\ \{\mathtt{X,x}\}=\mathtt{t}'_1\ \mathtt{in}\ \mathtt{t_2}} \qquad \text{(E-UNPACK)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t_2} : \{\mathtt{X} \mapsto \mathtt{U}\}\mathtt{T_2}}{\Gamma \vdash \{\exists\mathtt{X=U,t_2}\}\ \mathtt{as}\ \{\exists\mathtt{X,T_2}\} : \{\exists\mathtt{X,T_2}\}} \qquad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \exists\mathtt{X.T_{12}} \qquad \Gamma, \mathtt{X}, \mathtt{x}:\mathtt{T_{12}} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{let\ \{X,x\}=t_1\ in\ t_2} : \mathtt{T_2}} \qquad \text{(T-UNPACK)}$$

19.2 Data Abstraction with Existentials

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
```

19.2.1 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. □

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 14. Unlike the counters of Chapter 14, 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
```

19.2.2 Exercise: Implement `FlipFlop` objects with `Counter` objects as their internal representation type, following the model of the `FlipFlop` ADT in Section 19.2. □

Objects vs. ADTs

What we have seen in Section 19.2 falls significantly short of a full-blown model of object-oriented programming. Many of the features that we saw in Chapter 14, 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.

19.2.3 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? □

19.3 Encoding Existentials

The encoding of pairs as a polymorphic type in Exercise 18.4.5 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:

$$\texttt{\{}\exists\texttt{X,T\}} \stackrel{\text{def}}{=} \forall\texttt{Y. (}\forall\texttt{X. T}\to\texttt{Y)} \to \texttt{Y.}$$

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:

$$\texttt{\{}\exists\texttt{X=S,lt\} as \{}\exists\texttt{X,T\}} \stackrel{\text{def}}{=} \lambda\texttt{Y. }\lambda\texttt{f:(}\forall\texttt{X.T}\to\texttt{Y). ...}$$

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):

$$\texttt{\{}\exists\texttt{X=S,t\} as \{}\exists\texttt{X,T\}} \stackrel{\text{def}}{=} \lambda\texttt{Y. }\lambda\texttt{f:(}\forall\texttt{X.T}\to\texttt{Y). f [S] ...}$$

Now, the type application f [S] has type $\{X \mapsto S\}(T \to Y)$, i.e., $(\{X \mapsto S\}T) \to Y$. We can thus supply t (which, by rule T-PACK, has type $\{X \mapsto S\}T$) as the next argument:

$$\texttt{\{}\exists\texttt{X=S,t\} as \{}\exists\texttt{X,T\}} \stackrel{\text{def}}{=} \lambda\texttt{Y. }\lambda\texttt{f:(}\forall\texttt{X.T}\to\texttt{Y). f [S] t}$$

The type of the whole application f [S] t is now Y, as required.

To encode the unpacking construct

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \stackrel{\text{def}}{=} \ldots$$

we proceed as follows. First, the typing rule T-UNPACK tells us that t_1 should have some type $\{\exists X, T_{11}\}$, that t_2 should have type T_2 (under an extended context binding X and $x{:}T_{11}$), and that T_2 is the type we expect for the whole let...in... expression.[2] As in the Church encodings in Section 18.4, 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 t_1 and apply it to enough arguments to yield a result of the desired type T_2:

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \stackrel{\text{def}}{=} t_1 \ldots$$

The first argument to t_1 should be the desired result of the whole expression, i.e., T_2:

$$\texttt{let \{X,x\}=}t_1\texttt{ in }t_2 \stackrel{\text{def}}{=} t_1\ [T_2] \ldots$$

[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 $\mathtt{(\forall X.\ T{\to}T_2)\ \to\ T_2}$. That is, if we can now supply another argument of type $\mathtt{(\forall X.T{\to}T_2)}$, 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.

19.3.1 Exercise: What must we prove to show that our encoding of existentials is correct? □

19.3.2 Exercise [Recommended]: Take a blank piece of paper and, without looking at the above encoding, regenerate it from scratch. □

19.3.3 Exercise: Can universal types be encoded in terms of existential types? □

19.4 Implementation

```
type ty =
    ...
  | TySome of string * ty

type term =
    ...
  | TmPack of info * string * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

19.5 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 20

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:

$$\frac{\mathtt{S} <: \mathtt{T}}{\forall\mathtt{X.S} <: \forall\mathtt{X.T}}$$

We consider here 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.

20.1 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 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 12.

```
(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.

20.2 Definitions

We form System $F_{<:}$ by combining the types and terms of System F with the subtype relation from Chapter 12 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.

Kernel $F_{<:}$

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."

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVar)}$$

20.2.1 Exercise [Quick check]: Exhibit a subtyping derivation showing that

$$B<:Top,\ X<:B,\ Y<:X \vdash B\to Y <: X\to B.$$ □

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**.)

$$\frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{X} \mathtt{<:} \mathtt{T}} \qquad \text{(S-TVAR)}$$

For easy reference, here is the complete definition of kernel $F_{<:}$, with differences from previous systems highlighted:

$F^k_{<:}$: Bounded quantification $\rightarrow\ \forall\ \mathtt{<:}\ $ *bq*

Syntax

t ::=		*terms*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX<:T.t	*type abstraction*
	t [T]	*type application*
v ::=		*values*
	λx:T.t	*abstraction value*
	λX<:T.t	*type abstraction value*
T ::=		*types*
	X	*type variable*
	Top	*maximum type*
	T→T	*type of functions*
	∀X<:T.T	*universal type*
Γ ::=		*contexts*
	∅	*empty context*
	Γ, x:T	*term variable binding*
	Γ, X<:T	*type variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$(\lambda \mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_{12} \qquad \text{(E-BETA)}$$

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

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

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow \{X \mapsto T_2\}t_{12} \qquad \text{(E-BETA2)}$$

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

Subtyping $\boxed{\Gamma \vdash S <: T}$

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

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

$$\Gamma \vdash S <: \mathtt{Top} \qquad \text{(S-TOP)}$$

$$\frac{X{<:}T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVAR)}$$

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

$$\frac{\Gamma, X{<:}U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2} \qquad \text{(S-ALL)}$$

Typing $\boxed{\Gamma \vdash t : T}$

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

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 {\to} T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} {\to} T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

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

$$\frac{\Gamma, X{<:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{<:}T_1.t_2 : \forall X{<:}T_1.T_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}T_{12}} \qquad \text{(T-TAPP)}$$

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:

$$\frac{\mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\mathtt{U}\to\mathtt{S}_2 \mathrel{<:} \mathtt{U}\to\mathtt{T}_2}$$

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:

$F^f_{<:}$: "Full" bounded quantification $\to$ $\forall$ <: *bq* *full*

New subtyping rules $\boxed{\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T}_1 \mathrel{<:} \mathtt{S}_1 \qquad \Gamma, \mathtt{X}\mathtt{<:}\mathtt{T}_1 \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \forall\mathtt{X}\mathtt{<:}\mathtt{S}_1.\mathtt{S}_2 \mathrel{<:} \forall\mathtt{X}\mathtt{<:}\mathtt{T}_1.\mathtt{T}_2} \qquad \text{(S-ALL)}$$

Intuitively, the "full $F_{<:}$" quantifier subtyping rule can be understood as follows. A type $\mathtt{T} = \forall\mathtt{X}\mathtt{<:}\mathtt{T}_1.\mathtt{T}_2$ 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 $\mathtt{S} = \forall\mathtt{X}\mathtt{<:}\mathtt{S}_1.\mathtt{S}_2$, 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 $\mathtt{U} \mathrel{<:} \mathtt{T}_1$), 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.

20.2.2 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 $F_{<:}$. □

20.2.3 Exercise [Challenging]: Can you find any *useful* examples with this property? □

20.3 Examples

We now present some simple examples of programming in $F_{<:}$. More sophisticated uses of bounded quantification will appear in later chapters.

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 T_1 and 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

$$\frac{\Gamma \vdash \mathtt{S_1} \mathrel{<:} \mathtt{T_1} \qquad \Gamma \vdash \mathtt{S_2} \mathrel{<:} \mathtt{T_2}}{\Gamma \vdash \mathtt{Pair}\ \mathtt{S_1}\ \mathtt{S_2} \mathrel{<:} \mathtt{Pair}\ \mathtt{T_1}\ \mathtt{T_2}}$$

follows directly from the encoding.

20.3.1 Exercise [Quick check]: Show this. □

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:

20.3.2 Definition: For each $n \geq 0$ and types T_1 through 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 t_1 through 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:

$$\texttt{fst(}\underbrace{\texttt{snd(snd...(snd}}_{n-1\text{ times}}\texttt{t)...)} \qquad \square$$

From this abbreviation, we immediately obtain the following rules for subtyping and typing.

$$\frac{\Gamma \vdash {}^{i\in 1..n}\ \mathtt{S}_i \mathrel{<:} \mathtt{T}_i}{\Gamma \vdash \{\mathtt{S}_i{}^{i\in 1..n+k}\} \mathrel{<:} \{\mathtt{T}_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash {}^{i\in 1..n}\ \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{t}_i{}^{i\in 1..n}\} : \{\mathtt{T}_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash \mathtt{t} : \{\mathtt{T}_i{}^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t.i} : \mathtt{T}_i}$$

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:

20.3.3 Definition: Let L be a finite subset of $\mathcal{L}$ and let $\mathtt{S}_l$ be a type for each $l \in L$. Let m be the maximal index of any element of L, and

$$\hat{\mathtt{S}}_i = \begin{cases} \mathtt{S}_l & \text{if } \textit{label-with-index}(i) = l \in L \\ \mathtt{Top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record type $\{l\mathtt{:}\mathtt{S}_l{}^{l\in L}\}$ is defined as the flexible tuple $\{\hat{\mathtt{S}}_i{}^{i\in 1..m}\}$. Similarly, if $\mathtt{t}_l$ is a term for each $l : L$, then

$$\hat{\mathtt{t}}_i = \begin{cases} \mathtt{t}_i & \text{if } \textit{label-with-index}(i) = l \in L \\ \mathtt{top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record value $\{l\mathtt{=}\mathtt{t}_l{}^{l\in L}\}$ is $\{\hat{\mathtt{t}}_i{}^{i\in 1..m}\}$. The projection $\mathtt{t.l}_i$ is just the tuple projection `t.i`. $\square$

This encoding validates the expected rules for typing and subtyping:

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

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

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 18.4. 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 szero : SNat, sone : SNat, stwo : 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
```

20.3.4 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**.

20.3.5 Exercise [Recommended]: Generalize the type `CBool` of Church Booleans from Section 18.4 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 30, but these will require just a little more machinery, which we will develop in the intervening chapters.

20.4 Safety

We now consider the metatheory of both kernel and full systems of 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 $F^f_{<:}$.

The type preservation 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. (The soundness theorem belongs in the language definition, while decision procedures are buried in the compiler.) We develop the proof in detail for $F^k_{<:}$. The argument for $F^f_{<:}$ is very similar.

We begin with a couple of technical facts about the typing and subtyping relations. The proofs go by straightforward induction on derivations.

20.4.1 Lemma [Permutation]:

1. If $\Gamma \vdash \texttt{t} : \texttt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \texttt{t} : \texttt{T}$.
2. If $\Gamma \vdash \texttt{S} <: \texttt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \texttt{S} <: \texttt{T}$. □

20.4.2 Lemma [Weakening]:

1. If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{x} \notin dom(\Gamma)$, then $\Gamma, \texttt{x}{:}\texttt{S} \vdash \texttt{t} : \texttt{T}$.
2. If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{X} \notin dom(\Gamma)$, then $\Gamma, \texttt{X}{<:}\texttt{S} \vdash \texttt{t} : \texttt{T}$.
3. If $\Gamma \vdash \texttt{S} <: \texttt{T}$ and $\texttt{x} \notin dom(\Gamma)$, then $\Gamma, \texttt{x}{:}\texttt{S} \vdash \texttt{S} <: \texttt{T}$.
4. If $\Gamma \vdash \texttt{S} <: \texttt{T}$ and $\texttt{X} \notin dom(\Gamma)$, then $\Gamma, \texttt{X}{<:}\texttt{S} \vdash \texttt{S} <: \texttt{T}$. □

As usual, the proof of type preservation relies on several lemmas relating substitution with the typing and subtyping relations.

20.4.3 Definition: We write $\{\texttt{X} \mapsto \texttt{S}\}\Gamma$ for the context obtained by substituting S for X in the right-hand sides of all of the bindings in Γ. □

20.4.4 Exercise [Quick check]: Show the following properties of subtyping and typing derivations:

1. if $\Gamma, \texttt{X}{<:}\texttt{Q}, \Delta \vdash \texttt{S} <: \texttt{T}$ and $\Gamma \vdash \texttt{P} <: \texttt{Q}$, then $\Gamma, \texttt{X}{<:}\texttt{P}, \Delta \vdash \texttt{S} <: \texttt{T}$;
2. if $\Gamma, \texttt{X}{<:}\texttt{Q}, \Delta \vdash \texttt{t} : \texttt{T}$ and $\Gamma \vdash \texttt{P} <: \texttt{Q}$, then $\Gamma, \texttt{X}{<:}\texttt{P}, \Delta \vdash \texttt{t} : \texttt{T}$;

These properties are often called **narrowing** because they involve restricting the range of the variable X. □

Next, we have the usual lemma relating substitution and the typing relation.

20.4.5 Lemma [Substitution preserves typing]: If $\Gamma, \mathtt{x}{:}\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{q} : \mathtt{Q}$, then $\Gamma, \Delta \vdash \{\mathtt{x} \mapsto \mathtt{q}\}\mathtt{t} : \mathtt{T}$. □

Proof: Straightforward induction on a derivation of $\Gamma, \mathtt{x}{:}\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$, 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:

20.4.6 Lemma [Type substitution preserves subtyping]: If $\Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$. The only interesting cases are the last two:

Case S-TVAR: $\quad \mathtt{S} = \mathtt{Y} \qquad \mathtt{Y}{<:}\mathtt{T} \in (\Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta)(\mathtt{Y})$

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 $\mathtt{T} = \mathtt{Q}$ and $\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} = \mathtt{Q}$, and the result follows by S-REFL.

Case S-ALL: $\quad \mathtt{S} = \forall \mathtt{Z}{<:}\mathtt{U}_1.\mathtt{S}_2 \qquad \mathtt{T} = \forall \mathtt{Z}{<:}\mathtt{U}_1.\mathtt{T}_2$
$\qquad \Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta, \mathtt{Z}{<:}\mathtt{U}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta, \mathtt{Z}{<:}\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1 \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$. By S-ALL, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \forall \mathtt{Z}{<:}\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \forall \mathtt{Z}{<:}\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$, that is, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}{<:}\mathtt{U}_1.\mathtt{S}_2) <: \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}{<:}\mathtt{U}_1.\mathtt{T}_2)$, as required. □

20.4.7 Lemma [Type substitution preserves typing]: If $\Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$. We give just the interesting cases.

Case T-TAPP: $\quad \mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2] \qquad \Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{t}_1 : \forall \mathtt{Z}{<:}\mathtt{T}_{11}.\mathtt{T}_{12}$
$\qquad \mathtt{T} = \{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}{<:}\mathtt{T}_{11}.\mathtt{T}_{12})$, i.e, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \forall \mathtt{Z}{<:}\mathtt{T}_{11}.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12}$. By T-TAPP, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1\ [(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2])) : \{\mathtt{Z} \mapsto \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2\}(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12})$, i.e., $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\mathtt{t}_1\ [\mathtt{T}_2]) : \{\mathtt{X} \mapsto \mathtt{P}\}(\{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12})$.

Case T-SUB: $\quad \Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma, \mathtt{X}{<:}\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. By the preservation of subtyping under substitution (20.4.6), $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$, and the result follows by T-SUB. □

Next, we establish some simple structural facts about the subtype relation.

20.4.8 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 $\Gamma \vdash \mathtt{S} <: \forall \mathtt{X} <: \mathtt{U}_1 . \mathtt{T}_2$, then either $\mathtt{S}$ is a type variable or else $\mathtt{S} = \forall \mathtt{X} <: \mathtt{U}_1 . \mathtt{S}_2$ with $\Gamma, \mathtt{X} <: \mathtt{U}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2$. □

Proof: Part (1) follows by an easy induction on subtyping 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). □

20.4.9 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 $\Gamma \vdash \mathtt{S} <: \mathtt{T}$, where $\mathtt{X} <: \mathtt{S} \in \Gamma$.
4. If $\Gamma \vdash \mathtt{Top} <: \mathtt{T}$, then $\mathtt{T} = \mathtt{Top}$. □

We use Lemma 20.4.8 for one straightforward structural property of the typing relation that will be needed in the critical cases of the type preservation proof.

20.4.10 Lemma:

1. If $\Gamma \vdash \lambda \mathtt{x} : \mathtt{S}_1 . \mathtt{s}_2 : \mathtt{T}$ and $\Gamma \vdash \mathtt{T} <: \mathtt{U}_1 \to \mathtt{U}_2$, then $\Gamma \vdash \mathtt{U}_1 <: \mathtt{S}_1$ and there is some $\mathtt{S}_2$ such that $\Gamma, \mathtt{x} : \mathtt{S}_1 \vdash \mathtt{s}_2 : \mathtt{S}_2$ and $\Gamma \vdash \mathtt{S}_2 <: \mathtt{U}_2$.
2. If $\Gamma \vdash \lambda \mathtt{X} <: \mathtt{S}_1 . \mathtt{s}_2 : \mathtt{T}$ and $\Gamma \vdash \mathtt{T} <: \forall \mathtt{X} <: \mathtt{U}_1 . \mathtt{U}_2$, then $\mathtt{U}_1 = \mathtt{S}_1$ and there is some $\mathtt{S}_2$ such that $\Gamma, \mathtt{X} <: \mathtt{S}_1 \vdash \mathtt{s}_2 : \mathtt{S}_2$ and $\Gamma, \mathtt{X} <: \mathtt{S}_1 \vdash \mathtt{S}_2 <: \mathtt{U}_2$. □

Proof: Straightforward induction on typing derivations, using Lemma 20.4.8 for the induction case (rule T-SUB). □

With all these facts in-hand, the actual proof of type preservation is straightforward.

20.4.11 Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$. All of the cases are straightforward, using the facts established in the above lemmas.

Case T-VAR: $\mathtt{t} = \mathtt{x}$

This case cannot actually arise, since we assumed $\Gamma \vdash \mathtt{t} \longrightarrow \mathtt{t}'$ and there are no evaluation rules for variables.

Case T-ABS: $\mathtt{t} = \lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\mathtt{t}_2$

Ditto.

Case T-APP: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ $\quad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12}$
$\mathtt{T} = \mathtt{T}_{12}$ $\quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$

By the definition of the evaluation relation, there are three subcases to consider:

Subcase: $\Gamma \vdash \mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ $\quad \mathtt{t}' = \mathtt{t}_1'\ \mathtt{t}_2$

Then the result follows from the induction hypothesis and T-APP.

Subcase: $\mathtt{t}_1$ is a value $\quad \Gamma \vdash \mathtt{t}_2 \longrightarrow \mathtt{t}_2'$ $\quad \mathtt{t}' = \mathtt{t}_1\ \mathtt{t}_2'$

Similar.

Subcase: $\mathtt{t}_1 = \lambda\mathtt{x}\mathtt{:}\mathtt{U}_{11}\mathtt{.}\mathtt{u}_{12}$ $\quad \mathtt{t}' = \{\mathtt{x} \mapsto \mathtt{t}_2\}\mathtt{u}_{12}$

By Lemma 20.4.10, $\Gamma, \mathtt{x}\mathtt{:}\mathtt{U}_{11} \vdash \mathtt{u}_{12} : \mathtt{U}_{12}$ with $\Gamma \vdash \mathtt{T}_{11} \mathrel{<:} \mathtt{U}_{11}$ and $\Gamma \vdash \mathtt{U}_{12} \mathrel{<:} \mathtt{T}_{12}$. By the preservation of typing under substitution (Lemma 20.4.5), $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{t}_2\}\mathtt{u}_{12} : \mathtt{U}_{12}$, from which $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{t}_2\}\mathtt{u}_{12} : \mathtt{T}_{12}$ follows by T-SUB.

Case T-TABS: $\mathtt{t} = \lambda\mathtt{X}\mathtt{<:}\mathtt{U}\mathtt{.}\mathtt{t}$

Can't happen.

Case T-TAPP: $\mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2]$ $\quad \Gamma \vdash \mathtt{t} : \forall\mathtt{X}\mathtt{<:}\mathtt{T}_{11}\mathtt{.}\mathtt{T}_{12}$
$\mathtt{T} = \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}$ $\quad \Gamma \vdash \mathtt{T}_2 \mathrel{<:} \mathtt{T}_{11}$

By the definition of the evaluation relation, there are two subcases to consider:

Subcase: $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ $\quad \mathtt{t}' = \mathtt{t}_1'\ [\mathtt{T}_2]$

The result follows from the induction hypothesis and T-TAPP.

Subcase: $\mathtt{t}_1 = \lambda\mathtt{X}\mathtt{<:}\mathtt{U}_{11}\mathtt{.}\mathtt{u}_{12}$ $\quad \mathtt{t}' = \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{u}_{12}$

By Lemma 20.4.10, $\mathtt{U}_{11} = \mathtt{T}_{11}$ and $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{U}_{11} \vdash \mathtt{u}_{12} : \mathtt{U}_{12}$ with $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{U}_{11} \vdash \mathtt{U}_{12} \mathrel{<:} \mathtt{T}_{12}$. By the preservation of typing under substitution (20.4.5), $\Gamma \vdash \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{u}_{12} : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{U}_{12}$, from which $\Gamma \vdash \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{u}_{12} : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}$ follows by Lemma 20.4.6 and T-SUB.

Case T-SUB: $\Gamma \vdash \mathtt{t} : \mathtt{S}$ $\quad \Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$

By the induction hypothesis, $\Gamma \vdash \mathtt{t}' : \mathtt{S}$; the result follows by T-SUB. □

20.4.12 Exercise: Show how to extend the argument in this section to $F^{f}_{<:}$. □

20.5 Bounded Existential Types

This final section remains to be written.

Bounded existential quantification $F^{k}_{<:}$ + $\exists$

New syntactic forms

`T`	::=	...	*types*
		`{∃X<:T,T}`	*existential type*

New subtyping rules $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma, X{<:}U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X{<:}U, S_2\} <: \{\exists X{<:}U, T_2\}} \quad \text{(S-SOME)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : \{X \mapsto U\}T_2 \qquad \Gamma \vdash U <: T_1}{\Gamma \vdash \{\exists X{=}U, t_2\}\ \mathtt{as}\ \{\exists X{<:}T_1, T_2\} : \{\exists X{<:}T_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \exists X{<:}T_{11}.T_{12} \qquad \Gamma, X{<:}T_{11}, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ \{X,x\}{=}t_1\ \mathtt{in}\ t_2 : T_2} \quad \text{(T-UNPACK)}$$

20.5.1 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 Section 19.3. □

20.6 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 20.3) was considered by Ghelli thesis [Ghe90] and Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

An extension of $F_{<:}$ with intersection types was studied by Pierce [Pie91a, Pie97] and applied to the modeling of object-oriented languages with multiple inheritance by Compagnoni and Pierce [CP96].

Chapter 21

Implementing Bounded Quantification

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.

21.1 Promotion

In the typechecking algorithm for $\lambda_{<:}$ in Section 13.2, 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 take into account 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`, which 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`. Not too surprisingly, 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 $\Gamma \vdash S \Uparrow T$ to mean "`T` is the *least nonvariable supertype* of `S`," defined by repeated promotion of variables as follows:

Exposure

Exposure $\boxed{\Gamma \Vdash \mathtt{T} \Uparrow \mathtt{T}'}$

$$\frac{\mathtt{X}\texttt{<:}\mathtt{T} \in \Gamma \qquad \Gamma \Vdash \mathtt{T} \Uparrow \mathtt{T}'}{\Gamma \Vdash \mathtt{X} \Uparrow \mathtt{T}'} \qquad \text{(XA-PROMOTE)}$$

$$\frac{\mathtt{T} \text{ is not a type variable}}{\Gamma \Vdash \mathtt{T} \Uparrow \mathtt{T}} \qquad \text{(XA-OTHER)}$$

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.

21.1.1 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 $\mathtt{U}$ 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 $\Gamma \vdash \mathtt{S} <: \mathtt{U}$. □

21.2 Minimal Typing

The algorithm for calculating minimal types is built 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, and such a type will need to be promoted to its smallest non-variable supertype (its smallest **concrete** supertype, we might say) in order to be used on the left of an application or type application.

Algorithmic typing

Algorithmic typing $\boxed{\Gamma \Vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \Vdash \mathtt{x} : \mathtt{T}} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T_1} \Vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \Vdash \lambda \mathtt{x}:\mathtt{T_1}.\mathtt{t_2} : \mathtt{T_1} \to \mathtt{T_2}} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \Vdash \mathtt{t_1} : \mathtt{T_1} \qquad \Gamma \Vdash \mathtt{T_1} \Uparrow (\mathtt{T_{11}} \to \mathtt{T_{12}}) \qquad \Gamma \Vdash \mathtt{t_2} : \mathtt{T_2} \qquad \Gamma \Vdash \mathtt{T_2} <: \mathtt{T_{11}}}{\Gamma \Vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}}$$

(TA-APP)

$$\frac{\Gamma, \mathtt{X}<:\mathtt{T_1} \blacktriangleright \mathtt{t_2} : \mathtt{T_2}}{\Gamma \blacktriangleright \lambda\mathtt{X}<:\mathtt{T_1}.\mathtt{t_2} : \forall\mathtt{X}<:\mathtt{T_1}.\mathtt{T_2}} \quad \text{(TA-TABS)}$$

$$\frac{\Gamma \blacktriangleright \mathtt{t_1} : \mathtt{T_1} \qquad \Gamma \blacktriangleright \mathtt{T_1} \Uparrow \forall\mathtt{X}<:\mathtt{T_{11}}.\mathtt{T_{12}} \qquad \Gamma \blacktriangleright \mathtt{T_2} <: \mathtt{T_{11}}}{\Gamma \blacktriangleright \mathtt{t_1}\ [\mathtt{T_2}] : \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{T_{12}}} \quad \text{(TA-TAPP)}$$

The proofs of soundness and completeness of this algorithm with respect to the original typing rules are fairly routine.

21.2.1 Theorem [Minimal typing]:

1. If $\Gamma \blacktriangleright \mathtt{t} : \mathtt{T}$, then $\Gamma \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$, then $\Gamma \blacktriangleright \mathtt{t} : \mathtt{M}$ where $\Gamma \vdash \mathtt{M} <: \mathtt{T}$. □

Proof: Part (1) proceeds by a straightforward induction on algorithmic derivations. Part (2) is more interesting; it goes by induction on a derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$. (The most important cases are those for the rules T-APP and T-TAPP.)

Case T-VAR: $\mathtt{t} = \mathtt{x} \qquad \mathtt{x}:\mathtt{T} \in \Gamma$

By TA-VAR, $\Gamma \blacktriangleright \mathtt{x} : \mathtt{T}$. By S-REFL, $\Gamma \vdash \mathtt{T} <: \mathtt{T}$.

Case T-ABS: $\mathtt{t} = \lambda\mathtt{x}:\mathtt{T_1}.\mathtt{t_2} \qquad \Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2} \qquad \mathtt{T} = \mathtt{T_1}\to\mathtt{T_2}$

By the induction hypothesis, $\Gamma, \mathtt{x}:\mathtt{T_1} \blacktriangleright \mathtt{t_2} : \mathtt{M_2}$ for some $\mathtt{M_2}$ with $\Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{M_2} <: \mathtt{T_2}$, i.e. (since subtyping does not depend on term variable bindings), $\Gamma \vdash \mathtt{M_2} <: \mathtt{T_2}$. By TA-ABS, $\Gamma \vdash \mathtt{t} : \mathtt{T_1}\to\mathtt{M_2}$. Finally, by S-REFL and S-ARROW, we have $\Gamma \vdash \mathtt{T_1}\to\mathtt{M_2} <: \mathtt{T_1}\to\mathtt{T_2}$.

Case T-APP: $\mathtt{t} = \mathtt{t_1}\ \mathtt{t_2} \qquad \Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}}\to\mathtt{T_{12}}$
$\mathtt{T} = \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}$

By the induction hypothesis, we have $\Gamma \blacktriangleright \mathtt{t_1} : \mathtt{M_1}$ and $\Gamma \blacktriangleright \mathtt{t_2} : \mathtt{M_2}$, with $\Gamma \vdash \mathtt{M_1} <: \mathtt{T_{11}}\to\mathtt{T_{12}}$ and $\Gamma \vdash \mathtt{M_2} <: \mathtt{T_{11}}$. Let $\mathtt{N_1}$ be the least nonvariable supertype of $\mathtt{M_1}$—i.e., suppose $\Gamma \vdash \mathtt{M_1} \Uparrow \mathtt{N_1}$. By the promotion lemma (21.1.1), $\Gamma \vdash \mathtt{N_1} <: \mathtt{T_{11}}\to\mathtt{T_{12}}$. But we know that $\mathtt{N_1}$ is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $\mathtt{N_1} = \mathtt{N_{11}}\to\mathtt{N_{12}}$, with $\Gamma \vdash \mathtt{T_{11}} <: \mathtt{N_{11}}$ and $\Gamma \vdash \mathtt{N_{12}} <: \mathtt{T_{12}}$. By transitivity, $\Gamma \vdash \mathtt{M_2} <: \mathtt{N_{11}}$, so rule TA-APP applies and gives us $\Gamma \blacktriangleright \mathtt{t_1}\ \mathtt{t_2} : \mathtt{N_{12}}$, which satisfies the requirements.

Case T-TABS: $\mathtt{t} = \lambda\mathtt{X}<:\mathtt{T_1}.\mathtt{t_2} \qquad \Gamma, \mathtt{X}<:\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2} \qquad \mathtt{T} = \forall\mathtt{X}<:\mathtt{T_1}.\mathtt{T_2}$

By the induction hypothesis, $\Gamma, \mathtt{X}<:\mathtt{T_1} \blacktriangleright \mathtt{t_2} : \mathtt{M_2}$ for some $\mathtt{M_2}$ with $\Gamma, \mathtt{X}<:\mathtt{T_1} \vdash \mathtt{M_2} <: \mathtt{T_2}$. By TA-TABS, $\Gamma \blacktriangleright \mathtt{t} : \forall\mathtt{X}<:\mathtt{T_1}.\mathtt{M_2}$. Finally, by S-REFL and S-ALL, we have $\Gamma \vdash \forall\mathtt{X}<:\mathtt{T_1}.\mathtt{M_2} <: \forall\mathtt{X}<:\mathtt{T_1}.\mathtt{T_2}$.

Case T-TAPP: $\mathtt{t} = \mathtt{t_1}\ [\mathtt{T_2}]$ $\quad \Gamma \vdash \mathtt{t_1} : \forall\mathtt{X}\mathtt{<:}\mathtt{T_{11}}.\mathtt{T_{12}}$
$\mathtt{T} = \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{T_{12}}$ $\quad \Gamma \vdash \mathtt{T_2} \mathtt{<:} \mathtt{T_{11}}$

By the induction hypothesis, we have $\Gamma \blacktriangleright\!\!\!\vdash \mathtt{t_1} : \mathtt{M_1}$, with $\Gamma \vdash \mathtt{M_1} \mathtt{<:} \forall\mathtt{X}\mathtt{<:}\mathtt{T_{11}}.\mathtt{T_{12}}$. Let $\mathtt{N_1}$ be the least nonvariable supertype of $\mathtt{M_1}$—i.e., suppose $\Gamma \vdash \mathtt{M_1} \Uparrow \mathtt{N_1}$. By the promotion lemma (21.1.1), $\Gamma \vdash \mathtt{N_1} \mathtt{<:} \forall\mathtt{X}\mathtt{<:}\mathtt{T_{11}}.\mathtt{T_{12}}$. But we know that $\mathtt{N_1}$ is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $\mathtt{N_1} = \forall\mathtt{X}\mathtt{<:}\mathtt{N_{11}}.\mathtt{N_{12}}$, with $\mathtt{N_{11}} = \mathtt{T_{11}}$ and $\Gamma, \mathtt{X}\mathtt{<:}\mathtt{T_{11}} \vdash \mathtt{N_{12}} \mathtt{<:} \mathtt{T_{12}}$. Rule TA-TAPP gives us $\Gamma \blacktriangleright\!\!\!\vdash \mathtt{t_1}\ [\mathtt{T_2}] : \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{N_{12}}$, and the preservation of subtyping under substitution (20.4.6) yields $\Gamma \vdash \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{N_{12}} \mathtt{<:} \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{T_{12}} = \mathtt{T}$.

Case T-SUB: $\Gamma \vdash \mathtt{t} : \mathtt{S}$ $\quad \Gamma \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$

By the induction hypothesis, $\Gamma \blacktriangleright\!\!\!\vdash \mathtt{t} : \mathtt{M}$ with $\Gamma \vdash \mathtt{M} \mathtt{<:} \mathtt{S}$. By S-TRANS, $\Gamma \vdash \mathtt{M} \mathtt{<:} \mathtt{T}$, from which T-SUB yields the desired result. □

21.2.2 Corollary [Decidability of typing]: The $F^k_{<:}$ typing relation is decidable (if we are given a decision procedure for the subtyping relation). □

Proof: Given Γ and $\mathtt{t}$, we can check whether there is some $\mathtt{T}$ such that $\Gamma \vdash \mathtt{t} : \mathtt{T}$ by using the algorithmic typing rules to generate a proof of $\Gamma \blacktriangleright\!\!\!\vdash \mathtt{t} : \mathtt{T}$. If we succeed, then this $\mathtt{T}$ is also a type for $\mathtt{T}$ in the original typing relation (by part (1) of 21.2.1). If not, then part (2) of 21.2.1 implies that $\mathtt{t}$ 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 $\mathtt{t}$ when read from bottom to top. □

21.2.3 Exercise: Show how to add primitive booleans and conditionals to the minimal typing algorithm for $F^k_{<:}$. Solution on page 294. □

21.3 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 derivable subtyping statement $\Gamma \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$, we cannot decide which of the rules must have been used last in deriving it.

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 12: we remove the full reflexivity rule and replace it by a restricted reflexivity rule that applies only to type variables.

$$\Gamma \vdash \mathtt{X} <: \mathtt{X}$$

Next we must deal with S-TRANS. Unfortunately, unlike the simple subtyping relation studied in Chapter 12, the transitivity rule here interacts in an important way with another rule—namely S-TVAR, which allows assumptions about type variables to be used in deriving subtyping statements. For example, if

$$\Gamma \quad = \quad \mathtt{W<:Top},\ \mathtt{X<:W},\ \mathtt{Y<:X},\ \mathtt{Z<:Y}$$

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 observation can be made precise by introducing a new subtyping rule

$$\frac{\mathtt{X<:U} \in \Gamma \qquad \Gamma \vdash \mathtt{U} <: \mathtt{T}}{\Gamma \vdash \mathtt{X} <: \mathtt{T}}$$

that captures exactly this pattern of variable lookup followed by transitivity, and showing (as we will do below) that replacing the transitivity and variable rules by this one does not change the set of derivable subtyping statements.

These intuitions are summarized in the following definition. The algorithmic subtype relation of $F^k_{<:}$ is the least relation closed under the following rules:

Algorithmic subtyping

Algorithmic subtyping $\boxed{\Gamma \mapsto \mathtt{S} <: \mathtt{T}}$

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

$$\Gamma \mapsto \mathtt{X} <: \mathtt{X} \qquad \text{(SA-REFL-TVAR)}$$

$$\frac{\mathtt{X<:U} \in \Gamma \qquad \Gamma \mapsto \mathtt{U} <: \mathtt{T}}{\Gamma \mapsto \mathtt{X} <: \mathtt{T}} \qquad \text{(SA-TRANS-TVAR)}$$

$$\frac{\Gamma \Vdash \mathtt{T_1} \mathrel{<:} \mathtt{S_1} \qquad \Gamma \Vdash \mathtt{S_2} \mathrel{<:} \mathtt{T_2}}{\Gamma \Vdash \mathtt{S_1 \to S_2} \mathrel{<:} \mathtt{T_1 \to T_2}} \qquad \text{(SA-ARROW)}$$

$$\frac{\Gamma, \mathtt{X<:U_1} \Vdash \mathtt{S_2} \mathrel{<:} \mathtt{T_2}}{\Gamma \Vdash \forall \mathtt{X<:U_1.S_2} \mathrel{<:} \forall \mathtt{X<:U_1.T_2}} \qquad \text{(SA-ALL)}$$

21.3.1 Lemma [Reflexivity of the algorithmic subtype relation]: The statement $\Gamma \Vdash \mathtt{T} \mathrel{<:} \mathtt{T}$ is provable for all Γ and $\mathtt{T}$. □

Proof: Easy induction on $\mathtt{T}$. □

21.3.2 Lemma [Transitivity of the algorithmic subtype relation]: If $\Gamma \Vdash \mathtt{S} \mathrel{<:} \mathtt{Q}$ and $\Gamma \Vdash \mathtt{Q} \mathrel{<:} \mathtt{T}$, then $\Gamma \Vdash \mathtt{S} \mathrel{<:} \mathtt{T}$. □

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 $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{Top}$ by SA-TOP. Moreover, if the left-hand derivation is an instance of SA-TOP, then $\mathtt{Q} = \mathtt{Top}$ 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 $\mathtt{S} = \mathtt{X}$ with $\mathtt{X<:U} \in \Gamma$ and we have a subderivation with conclusion $\Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{Q}$. By the induction hypothesis, $\Gamma \Vdash \mathtt{U} \mathrel{<:} \mathtt{T}$, and, by SA-TRANS-TVAR again, $\Gamma \Vdash \mathtt{X} \mathrel{<:} \mathtt{T}$, as required.

If the left-hand derivation ends with an instance of SA-ARROW, then we have $\mathtt{S} = \mathtt{S_1 \to S_2}$ and $\mathtt{Q} = \mathtt{Q_1 \to Q_2}$, with subderivations $\Gamma \Vdash \mathtt{Q_1} \mathrel{<:} \mathtt{S_1}$ and $\Gamma \Vdash \mathtt{S_2} \mathrel{<:} \mathtt{Q_2}$. 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 $\mathtt{T} = \mathtt{T_1 \to T_2}$, and two more subderivations $\Gamma \Vdash \mathtt{T_1} \mathrel{<:} \mathtt{Q_1}$ and $\Gamma \Vdash \mathtt{Q_2} \mathrel{<:} \mathtt{T_2}$. We now apply the induction hypothesis twice, obtaining $\Gamma \Vdash \mathtt{T_1} \mathrel{<:} \mathtt{S_1}$ and $\Gamma \Vdash \mathtt{S_2} \mathrel{<:} \mathtt{T_2}$. Finally, SA-ARROW yields $\Gamma \Vdash \mathtt{S_1 \to S_2} \mathrel{<:} \mathtt{T_1 \to T_2}$, as required.

The case where the left-hand derivation ends with an instance of SA-ALL is similar. □

21.3.3 Theorem [Soundness and completeness of algorithmic subtyping]:

1. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ then $\Gamma \Vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.
2. If $\Gamma \Vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ then $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$. □

Proof: Both directions proceed by induction on derivations. Soundness is routine. 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 21.3.1 and 21.3.2. □

Finally, we should check that the subtyping rules define an algorithm that is *total*—i.e., that always terminates no matter what input it is given. 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.

21.3.4 Definition: The **weight** of a type $\mathtt{T}$ in a context Γ, written $weight_{\Gamma}(\mathtt{T})$, is defined as follows:

$$\begin{array}{lcll}
weight_{\Gamma}(\mathtt{X}) & = & weight_{\Gamma_1}(\mathtt{U}) + 1 & \text{if } \Gamma = \Gamma_1,\, \mathtt{X<:U},\, \Gamma_2 \\
weight_{\Gamma}(\mathtt{Top}) & = & 1 & \\
weight_{\Gamma}(\mathtt{T_1 \to T_2}) & = & weight_{\Gamma}(\mathtt{T_1}) + weight_{\Gamma}(\mathtt{T_2}) + 1 & \\
weight_{\Gamma}(\forall \mathtt{X<:T_1.T}) & = & weight_{\Gamma,\,\mathtt{X<:T_1}}(\mathtt{T_2}) + 1 &
\end{array}$$

The **weight** of a subtyping statement "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$" is the maximum weight of $\mathtt{S}$ and $\mathtt{T}$ in Γ. □

21.3.5 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. □

21.4 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:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \forall \mathtt{X<:S_1.S_2} <: \forall \mathtt{X<:T_1.T_2}} \qquad \text{(S-ALL)}$$

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:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \mapsto \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \mapsto \forall \mathtt{X<:S_1.S_2} <: \forall \mathtt{X<:T_1.T_2}} \qquad \text{(SA-ALL)}$$

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.

21.4.1 Lemma [Transitivity and narrowing]:

1. If $\Gamma \mapsto$ `S <: Q` and $\Gamma \mapsto$ `Q <: T`, then $\Gamma \mapsto$ `S <: T`.

2. If Γ, `X<:Q`, $\Delta \mapsto$ `M <: N` and $\Gamma \mapsto$ `P <: Q` then Γ, `X<:P`, $\Delta \mapsto$ `S <: T`. □

Proof: The two parts are proved simultaneously, by induction on the size of `Q`. At each stage of the induction, the argument for part (2) assumes that part (1) has been established already for the `Q` in question; part (1), on the other hand, uses part (2) only for strictly smaller `Q`s.

1. Most of the argument is identical to the proof of 21.3.2. The only difference lies in the case where both of the given derivations end in instances of SA-ALL. In this case, we have

   $S = \forall X{<:}S_1.S_2$
   $Q = \forall X{<:}Q_1.Q_2$
   $T = \forall X{<:}T_1.T_2$
   $\Gamma \vdash Q_1 <: S_1$
   $\Gamma, X{<:}Q_1 \mapsto S_2 <: Q_2 \Gamma \vdash T_1 <: Q_1$
   $\Gamma, X{<:}T_1 \mapsto Q_2 <: S_2$

   as subderivations. By part (1) of the induction hypothesis, we can immediately combine the three subderivations for the bounds to obtain $\Gamma \mapsto T_1 <: S_1$. 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 $\Gamma, X{<:}Q_1 \mapsto S_2 <: Q_2$, obtaining $\Gamma, X{<:}T_1 \mapsto S_2 <: Q_2$. Now part (1) of the induction hypothesis applies, yielding $\Gamma, X{<:}T_1 \mapsto S_2 <: T_2$. Finally, by SA-ALL, $\Gamma \mapsto \forall X{<:}S_1.S_2 <: \forall X{<:}T_1.T_2$, as required.

2. Given `Q` of a certain size, we use an inner induction on the size of a derivation of Γ, `X<:Q`, $\Delta \mapsto$ `S <: T`, with a case analysis on the last rule used in this derivation.

   The only interesting case is SA-TRANS-TVAR, where `S` = `X` and Γ, `X<:Q`, $\Delta \mapsto$ `Q <: T` as a subderivation. By the inner induction hypothesis, Γ, `X<:P`, $\Delta \mapsto$ `Q <: T`. Rule SA-TRANS-TVAR now yields Γ, `X<:P`, $\Delta \mapsto$ `X <: T`, as required. □

21.4.2 Exercise: Show that $F^k_{<:}$ has joins and bounded meets. Solution on page 294. □

21.4.3 Exercise: Consider the types (due to Ghelli [Ghe90, p. 92])

$S = \forall X{<:}Y{\to}Z.Y{\to}Z$

and

$$\mathtt{T} = \forall\mathtt{X}\mathtt{<:}\mathtt{Y}'\to\mathtt{Z}'.\mathtt{Y}'\to\mathtt{Z}'$$

and the context $\Gamma = \mathtt{Y<:Top}, \mathtt{Z<:Top}, \mathtt{Y'<:Y}, \mathtt{Z'<:Z}$.

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 298. □

21.5 Undecidability of Subtyping in Full $F_{<:}$

Unfortunately, the proof of termination of the $F^k_{<:}$ subtyping algorithm in Section **??** does not work for $F^f_{<:}$.

21.5.1 Exercise [Quick check]: Why not? □

In fact, not only does this particular proof technique not work—the subtyping algorithm actually does *not* terminate on some inputs. Here is an example, due to Ghelli [Ghe95], that makes the algorithm diverge. First define the following abbreviation:

$$\neg\mathtt{S} \stackrel{\text{def}}{=} \forall\mathtt{X<:S.X}.$$

21.5.2 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:

$$\mathtt{T} \quad = \quad \forall\mathtt{X<:Top.}\ \neg(\forall\mathtt{Y<:X}.\neg\mathtt{Y}).$$

If we use the algorithmic subtyping rules bottom-to-top to attempt to construct a subtyping derivation for the statement

$$\mathtt{X_0<:T} \vdash \mathtt{X_0} <: (\forall\mathtt{X_1<:X_0}.\neg\mathtt{X_1})$$

we end up in an infinite regress of larger and larger "subgoals":

$$\begin{array}{llll}
\mathtt{X_0<:T} & \vdash & \mathtt{X_0} & <: & (\forall\mathtt{X_1<:X_0}.\neg\mathtt{X_1}) \\
\mathtt{X_0<:T} & \vdash & \forall\mathtt{X_1<:Top.}\ \neg(\forall\mathtt{X_2<:X_1}.\neg\mathtt{X_2}) & <: & (\forall\mathtt{X_1<:X_0}.\neg\mathtt{X_1}) \\
\mathtt{X_0<:T}, \mathtt{X_1<:X_0} & \vdash & \neg(\forall\mathtt{X_2<:X_1}.\neg\mathtt{X_2}) & <: & \neg\mathtt{X_1} \\
\mathtt{X_0<:T}, \mathtt{X_1<:X_0} & \vdash & \mathtt{X_1} & <: & (\forall\mathtt{X_2<:X_1}.\neg\mathtt{X_2}) \\
\mathtt{X_0<:T}, \mathtt{X_1<:X_0} & \vdash & \mathtt{X_0} & <: & (\forall\mathtt{X_2<:X_1}.\neg\mathtt{X_2})
\end{array}$$

etc.

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 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.

21.5.3 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.

21.5.4 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 (pretty horrible) type

$$\begin{aligned}\mathtt{T} = {} & \forall \mathtt{X}_0\mathtt{<:Top}.\forall \mathtt{X}_1\mathtt{<:Top}.\forall \mathtt{X}_2\mathtt{<:Top}.\\ & \quad \neg(\forall \mathtt{Y}_0\mathtt{<:X}_0.\forall \mathtt{Y}_1\mathtt{<:X}_1.\forall \mathtt{Y}_2\mathtt{<:X}_2.\neg \mathtt{X}_0)\end{aligned}$$

and consider the subtyping statement

$$\begin{array}{ll} \vdash & \mathtt{T} \\ \mathtt{<:} & \forall \mathtt{X}_0\mathtt{<:T}.\forall \mathtt{X}_1\mathtt{<:P}.\forall \mathtt{X}_2\mathtt{<:Q}. \\ & \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. \\ & \quad\quad \neg(\forall \mathtt{Z}_0\mathtt{<:Y}_0.\forall \mathtt{Z}_1\mathtt{<:Y}_2.\forall \mathtt{Z}_2\mathtt{<:Y}_1.\ \mathtt{U})). \end{array}$$

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.)

$$\begin{array}{llll}
 & \vdash & \mathtt{T} & \\
 & \mathtt{<:} & \forall \mathtt{X}_0\mathtt{<:T}.\forall \mathtt{X}_1\mathtt{<:}\boxed{\mathtt{P}}.\forall \mathtt{X}_2\mathtt{<:}\boxed{\mathtt{Q}}. & \\
 & & \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \\
 & & \quad\quad \neg(\forall \mathtt{Z}_0\mathtt{<:Y}_0.\forall \mathtt{Z}_1\mathtt{<:Y}_2.\forall \mathtt{Z}_2\mathtt{<:Y}_1.\ \mathtt{U})) & \\
\text{iff} & \vdash & \neg(\forall \mathtt{Y}_0\mathtt{<:T}.\forall \mathtt{Y}_1\mathtt{<:}\boxed{\mathtt{P}}.\forall \mathtt{Y}_2\mathtt{<:}\boxed{\mathtt{Q}}.\neg \mathtt{T}) & \\
 & \mathtt{<:} & \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \text{by Fact 21.5.4} \\
 & & \quad \neg(\forall \mathtt{Z}_0\mathtt{<:Y}_0.\forall \mathtt{Z}_1\mathtt{<:Y}_2.\forall \mathtt{Z}_2\mathtt{<:Y}_1.\ \mathtt{U})) & \\
\text{iff} & \vdash & (\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. & \\
 & & \quad \neg(\forall \mathtt{Z}_0\mathtt{<:Y}_0.\forall \mathtt{Z}_1\mathtt{<:Y}_2.\forall \mathtt{Z}_2\mathtt{<:Y}_1.\ \mathtt{U})) & \\
 & \mathtt{<:} & (\forall \mathtt{Y}_0\mathtt{<:T}.\forall \mathtt{Y}_1\mathtt{<:}\boxed{\mathtt{P}}.\forall \mathtt{Y}_2\mathtt{<:}\boxed{\mathtt{Q}}.\neg \mathtt{T}) & \text{by Fact 21.5.2} \\
\text{iff} & \vdash & \neg(\forall \mathtt{Z}_0\mathtt{<:T}.\forall \mathtt{Z}_1\mathtt{<:}\boxed{\mathtt{Q}}.\forall \mathtt{Z}_2\mathtt{<:}\boxed{\mathtt{P}}.\ \mathtt{U})) & \\
 & \mathtt{<:} & \neg \mathtt{T} & \text{by Fact 21.5.4} \\
\text{iff} & \vdash & \mathtt{T} & \\
 & \mathtt{<:} & (\forall \mathtt{Z}_0\mathtt{<:T}.\forall \mathtt{Z}_1\mathtt{<:}\boxed{\mathtt{Q}}.\forall \mathtt{Z}_2\mathtt{<:}\boxed{\mathtt{P}}.\ \mathtt{U})) & \text{by Fact 21.5.2}
\end{array}$$

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

$$\mathtt{U} \quad = \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. \\ \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_2.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_1.\ \mathtt{U}'))$$

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

$$\mathtt{U} \quad = \quad \neg(\forall \mathtt{Y}_0\mathtt{<:Top}.\forall \mathtt{Y}_1\mathtt{<:Top}.\forall \mathtt{Y}_2\mathtt{<:Top}. \\ \neg(\forall \mathtt{Z}_0\mathtt{<:}\mathtt{Y}_0.\forall \mathtt{Z}_1\mathtt{<:}\mathtt{Y}_1.\forall \mathtt{Z}_2\mathtt{<:}\mathtt{Y}_2.\ \mathtt{Y}_1))$$

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. Conditional 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 counters, each holding a natural number (cf. [HU79], for example)—to subtyping statements.

21.5.5 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.

21.5.6 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.?

Hint on page 298. □

Chapter 22

Denotational Semantics

By Martin Hofmann and Benjamin Pierce

Some notation is out of date in this chapter.

Denotational semantics is a compositional assigment of mathematical objects to program phrases (terms, types, and typing and subtyping statements). Compositionality means that the meaning of a phrase is defined in terms of its outermost constructor and the meanings of its immediate subphrases.

The foremost and easiest application of semantics is demonstrating the soundness and consistency of equational theories—in particular, the one generated by the reduction rules. (To show that the equational theory is sound, we show that terms equated by the theory have equal interpretations; to show that it is consistent—i.e., that not all terms are equated by the equational theory—it suffices to exhibit two terms with different interpretations.) Semantic models can also suggest extensions to the syntax, in particular program logics.

If a semantics is "computationally adequate" (we'll come back to this term later on), it can also be used to reason about observational equivalence of programs. For example, it is intuitively the case that the two records `p` $=$ `{x=0,y=0}` and `q` $=$ `{x=0,y=1}` are equal when used at type `{|x:Nat|}`, because the only thing that a program can do with a record of type `{|x:Nat|}` is to project out its `x` field, and `p.x` $=$ `q.x` $=$ `0`. The semantics we are going to develop will allow us to argue that, indeed, `p` and `q` are observationally equivalent at type `{|x:Nat|}`, i.e., that for every closed term `c : {|x:Nat|}`$\to$`Nat`, the terms `c p` and `c q` of type `Nat` yield the same result.

Finally, in some cases a (non-standard) semantics can be employed to establish syntactic meta-properties such as normalization or confluence.

22.1 Types as Subsets of the Natural Numbers

In building semantic models of programming languages, types are commonly modeled as sets (or, if the language includes recursion, "domains"); closed terms are modeled as elements of the appropriate sets, open terms as functions of the interpretations of their free variables. For example, a straightforward model of the simply typed lambda-calculus can be obtained by interpreting base types as arbitrary sets (e.g. $[\![\mathtt{Nat}]\!] = \mathbb{N}$) and each function type $\mathtt{S}\rightarrow\mathtt{T}$ as the set of functions from $[\![\mathtt{S}]\!]$ to $[\![\mathtt{T}]\!]$.

For $F_{<:}$, a more refined framework is required, due principally to the impredicativity of the universal quantifier in System F—the fact that the bound type variable in a type $\forall$[X]T ranges over all types, including $\forall$[X]T itself. Accordingly, in a simple set-theoretic model, the first candidate for an intepretation of a universal type would be a function whose domain is the set of all sets; such a function cannot exist, for cardinality reasons. A more refined strategy would be to restrict the domain of polymorphic functions to only those sets which actually arise as interpretations of types in our programming language. However, Reynolds [Rey84, RP] has shown that, no matter how we interpret universal types, there cannot exist a model of System F in which a function type $\mathtt{S}\rightarrow\mathtt{T}$ is interpreted as the full set of functions from $[\![\mathtt{S}]\!]$ to $[\![\mathtt{T}]\!]$. Intuitively, what Reynolds shows is that there is a type[1] R in System F whose interpretation must be the same size as the function space (R$\rightarrow$Bool)$\rightarrow$Bool; the fact that no such set exists is a direct consequence of Russell's Paradox.

The key to a successful interpretation of $F_{<:}$ lies in the observation that the functions we are interested in always arise as the interpretations of finite expressions in our programming language—they are "algorithms," not arbitrary set-theoretic functions. This means that we may restrict the interpretations of types to subsets of some *a priori* given universe of data values and algorithms; universal quantifiers may now be taken to range over the subsets of this set.

The Universe of Natural Numbers

A convenient choice for the universe of data values and algorithms is the set $\mathbb{N}$ of natural numbers. As is well known, all flat datatypes such as integers, strings, lists, floating point numbers, etc. can be encoded as natural numbers.[2]

We will not need to be too formal, here, about the precise encoding of algorithms as numbers. We just assume that

1. we are given some way of intepreting a number n as a partial function $\{n\}$ from numbers to numbers, and that

[1]Namely R = $\forall$[X] (((X$\rightarrow$Bool)$\rightarrow$Bool) $\rightarrow$ X) $\rightarrow$ X.

[2]For example, a pair (m, n) of numbers can be encoded as $2^m \cdot (2 \cdot n + 1)$.

2. every "intuitively computable partial function" f on the natural numbers is represented by some number n, in the sense that $\{n\}(m) = f(m)$ (where by the equality symbol here we mean the so-called "Kleene equality": either both $\{n\}(m)$ and $f(m)$ are undefined, or both are defined and they are equal).

We write $\underline{\lambda}x.\ g(x)$ for the number representing the function mapping each x to $g(x)$, where $g(x)$ is some concrete description of a computable function.

Subset Semantics for $F_{<:}$

We can obtain a simple semantics for $F_{<:}$ in this setting by interpreting types as subsets of $\mathbb{N}$ and terms as elements of the meaning of their types. More formally, the interpretation of simple types (i.e., the types of the simply typed lambda-calculus) is given by the following recursive definition:

$$\begin{array}{lcl}
[\![\mathtt{Nat}]\!] & = & \mathbb{N} \\
[\![\mathtt{Top}]\!] & = & \mathbb{N} \\
[\![\mathtt{S}{\rightarrow}\mathtt{T}]\!] & = & \{t \mid \forall x \in [\![\mathtt{S}]\!].\ \{t\}(x) \in [\![\mathtt{T}]\!]\} \\
[\![\{\!|\mathtt{l}_1{:}\mathtt{T}_1 \ldots \mathtt{l}_n{:}\mathtt{T}_n|\!\}]\!] & = & \{t \mid \forall \mathtt{l}_i \in \{\mathtt{l}_1 \ldots \mathtt{l}_n\}.\ \{t\}(\mathtt{l}_i) \in [\![\mathtt{T}_i]\!]\}.
\end{array}$$

A few comments about this definition are in order:

- By convention, a statement like $\{t\}(x) \in [\![\mathtt{T}]\!]$ means, in particular, that $\{t\}(x)$ is defined. Thus, if $t \in [\![\mathtt{S}{\rightarrow}\mathtt{T}]\!]$, then $\{t\}$ is defined whenever it is applied to an element of $[\![\mathtt{S}]\!]$.

  In other words, although we are interpreting the elements of $\mathbb{N}$ as partial functions, the interpretations of types turn out to be functions that are total on the elements of their domains.

- For notational convenience, we assume that record labels are drawn from $\mathbb{N}$, so that records can be modeled as partial functions on numbers and field projection as application.

In order to interpret types containing variables, we must parameterize the above definition with an **environment** mapping type variables to subsets of $\mathbb{N}$ and (for subsequent use) term variables to natural numbers. We parameterize the above definition by an environment ρ

$$\begin{array}{lcl}
[\![\mathtt{Nat}]\!]_\rho & = & \mathbb{N} \\
[\![\mathtt{Top}]\!]_\rho & = & \mathbb{N} \\
[\![\mathtt{S}{\rightarrow}\mathtt{T}]\!]_\rho & = & \{t \mid \forall x \in [\![\mathtt{S}]\!]_\rho.\ \{t\}(x) \in [\![\mathtt{T}]\!]_\rho\} \\
[\![\{\!|\mathtt{l}_1{:}\mathtt{T}_1 \ldots \mathtt{l}_n{:}\mathtt{T}_n|\!\}]\!]_\rho & = & \{t \mid \forall \mathtt{l}_i \in \{\mathtt{l}_1 \ldots \mathtt{l}_n\}.\ \{t\}(\mathtt{l}_i) \in [\![\mathtt{T}_i]\!]_\rho\}
\end{array}$$

and add appropriate clauses for type variables and universal types:

$$\begin{array}{lcl}
[\![\mathtt{X}]\!]_\rho & = & \rho(\mathtt{X}) \\
[\![\forall[\mathtt{X}{<:}\mathtt{S}]\mathtt{T}]\!]_\rho & = & \{t \mid \forall U \subseteq [\![\mathtt{S}]\!]_\rho.\ t \in [\![\mathtt{T}]\!]_{\rho+\{\mathtt{X}\mapsto U\}}\} \\
 & & \text{that is, } \bigcap_{U \subseteq [\![\mathtt{S}]\!]_\rho} [\![\mathtt{T}]\!]_{\rho+\{\mathtt{X}\mapsto U\}}
\end{array}$$

Similarly we define the meaning of terms:

$$
\begin{array}{lcl}
[\![\mathtt{x}]\!]_\rho & = & \rho(\mathtt{x}) \\
[\![\mathtt{0}]\!]_\rho & = & 0 \\
[\![\mathtt{succ\ t}]\!]_\rho & = & [\![\mathtt{t}]\!]_\rho + 1 \\
[\![\mathtt{iter\ T\ t_1\ t_2\ t_3}]\!]_\rho & = & [\![\mathtt{t_2}]\!]_\rho^n([\![\mathtt{t_3}]\!]_\rho),\ \text{where } n = [\![\mathtt{t_1}]\!]_\rho \\
[\![\mathtt{t.l}]\!]_\rho & = & \{[\![\mathtt{t}]\!]_\rho\}(\mathtt{l}) \\
[\![\mathtt{\{l_i{=}t_i\}}]\!]_\rho & = & \underline{\lambda} l.\ \text{if } l = \mathtt{l_i} \text{ then } [\![\mathtt{t_i}]\!]_\rho \text{ else undefined} \\
[\![\mathtt{t\ t'}]\!]_\rho & = & \{[\![\mathtt{t}]\!]_\rho\}([\![\mathtt{t'}]\!]_\rho) \\
[\![\mathtt{fun[x{:}S]t}]\!]_\rho & = & \underline{\lambda} v.\ [\![\mathtt{t}]\!]_{\rho+\{\mathtt{x}\mapsto v\}} \\
[\![\mathtt{fun[X{<:}S]t}]\!]_\rho & = & [\![\mathtt{t}]\!]_\rho \\
[\![\mathtt{t\ S}]\!]_\rho & = & [\![\mathtt{t}]\!]_\rho
\end{array}
$$

Notice that the meaning $[\![\mathtt{t}]\!]_\rho$ only depends on the restriction of ρ to term variables; the interpretation of types, even of those which occur as annotations in `t`, is irrelevant.

Notice also that the semantics of a term may be undefined either because one of its variables is not declared in the environment or because one of the semantic applications is undefined. Our aim is to formulate a soundness theorem which states that the meaning of every well-typed term is defined and is contained in the meaning of its type. To this end, we need to define what it means for an environment to match a typing context.

22.1.1 Definition: Let ρ be an environment and Γ be a context. Say that ρ satisfies Γ (written $\rho \models \Gamma$) if, whenever `X<:S` occurs in Γ, we have $\rho(\mathtt{X})$ and $[\![\mathtt{S}]\!]_\rho$ both defined and the former a subset of the latter and, similarly, whenever `x:S` occurs in Γ, we have $\rho(\mathtt{x})$ and $[\![\mathtt{S}]\!]_\rho$ both defined with the former an element of the latter. □

22.1.2 Theorem [Soundness]: If Γ is a context and ρ an environment satisfying Γ, then:

1. If all the free type variables in `S` appear in $dom(\Gamma)$ then $[\![\mathtt{S}]\!]_\rho$ is defined.
2. If $\Gamma \vdash \mathtt{S{<:}T}$ then $[\![\mathtt{S}]\!]_\rho \subseteq [\![\mathtt{T}]\!]_\rho$.
3. If $\Gamma \vdash \mathtt{t} \in \mathtt{S}$ then $[\![\mathtt{t}]\!]_\rho$ is an element of $[\![\mathtt{S}]\!]_\rho$. □

The proof is by induction on derivations. The most important ingredient is the following substitution lemma.

22.1.3 Lemma [Semantic substitution]:

1. If $[\![\mathtt{S}]\!]_{\rho+\{\mathtt{X}\mapsto[\![\mathtt{T}]\!]_\rho\}}$ is defined, then so is $[\![\{\mathtt{X} \mapsto \mathtt{T}\}\mathtt{S}]\!]_\rho$ and they are equal.
2. If $[\![\mathtt{t}]\!]_{\rho+\{\mathtt{x}\mapsto[\![\mathtt{t'}]\!]_\rho\}}$ is defined, then so is $[\![\{\mathtt{x} \mapsto \mathtt{t'}\}\mathtt{t}]\!]_\rho$ and they are equal. □

Proof: By induction on types (part 1) or terms (part 2). □

Notice that this substitution lemma is nothing but a formalisation of the requirement of *compositionality* mentioned in the introduction to this chapter.

Proof of Theorem 22.1.2: Part (1) is by induction on types. Part (2) is by induction on subtyping derivations (not necessarily algorithmic derivations: the transitivity rule is not problematic). Part (3) is by induction on typing derivations. None of the cases in these inductions are difficult; as examples, we show the cases for function abstraction and type application in part (3).

For the function abstraction case, suppose that $\Gamma, \texttt{x:S} \vdash \texttt{t : T}$ and that $\rho \models \Gamma$. Then $[\![\texttt{fun[x:S]t}]\!]_\rho = \underline{\lambda}v.\ [\![\texttt{t}]\!]_{\rho+\{\texttt{x}\mapsto v\}}$; call this number d. We must show $d \in [\![\texttt{S}\to\texttt{T}]\!]_\rho$—that is, if $a \in [\![\texttt{S}]\!]_\rho$, then $\{d\}(a)$ is defined and belongs to $[\![\texttt{T}]\!]_\rho$. Now, if $a \in [\![\texttt{S}]\!]_\rho$, then $\{d\}(a) = [\![\texttt{t}]\!]_{\rho+\{\texttt{x}\mapsto a\}}$. Since $a \in [\![\texttt{S}]\!]_\rho$, the extended environment $\rho + \{\texttt{x} \mapsto a\}$ satisfies $\Gamma, \texttt{x:S}$, so the induction hypothesis applies, yielding $\{d\}(a) \in [\![\texttt{T}]\!]_\rho$, as required.

For the type application case, suppose that $\Gamma \vdash \texttt{t : }\forall\texttt{[X<:S]T}$, that $\Gamma \vdash \texttt{U <: S}$, and that $\rho \models \Gamma$. Part (2) gives us $[\![\texttt{U}]\!]_\rho \subseteq [\![\texttt{S}]\!]_\rho$. Using the induction hypothesis on $\texttt{t}$ and the interpretation of universal types, we obtain $[\![\texttt{t}]\!]_\rho \in [\![\texttt{T}]\!]_{\rho+\{\texttt{X}\mapsto[\![\texttt{U}]\!]_\rho\}}$. Thus, by Lemma 22.1.3 and the interpretation of type application, we have $[\![\texttt{t U}]\!]_\rho = [\![\texttt{t}]\!]_\rho \in [\![\{\texttt{X} \mapsto \texttt{U}\}\texttt{T}]\!]_\rho$. □

22.1.4 Exercise [Quick check]: How can it happen that $[\![\{\texttt{x} \mapsto \texttt{t}'\}\texttt{t}]\!]_\rho$ is defined, but $[\![\texttt{t}]\!]_{\rho+\{\texttt{x}\mapsto[\![\texttt{t}']\!]_\rho\}}$ is not? □

22.1.5 Exercise: Show the proofs of the cases for record and quantifier subtyping for part (2) of the last proof. □

An application of the subset model is that it can be used to decide whether certain types are inhabited by any closed terms. For example, $[\![\forall\texttt{[X<:Top]X}]\!] = \bigcap_{U\subseteq\mathbb{N}} U$ is the empty set[3]; so there can be no closed term of type $\forall\texttt{[X<:Top]X}$, because its interpretation would have to be an element of the empty set. Similarly, the set $[\![\forall\texttt{[X<:Top]}\forall\texttt{[Y<:Top]((X}\to\texttt{Y)}\to\texttt{(X}\to\texttt{Y))}\to\texttt{X}\to\texttt{Y}]\!]$—the interpretation of the type of the fixed-point combinator—is empty, which shows that the fixed-point combinator cannot be interpreted in the subset model, and hence cannot be defined in pure System $F_{<:}$. This ties in with the observation made above that all functions in the subset model are total.

[3]From here on, when the environment ρ in an expression $[\![\texttt{T}]\!]_\rho$ is empty, we will usually omit it.

Problems with the Subset Semantics

Though it validates the typing, subtyping, and evaluation relations, the subset semantics is unsatisfactory because it does not validate some of the rules of the reduction relation. More precisely, from

$$[\![\texttt{t}]\!]_{\rho+\{x\mapsto v\}} = [\![\texttt{t}']\!]_{\rho+\{x\mapsto v\}} \quad \text{for all } v \in [\![\texttt{S}]\!]_\rho$$

we cannot conclude that

$$[\![\texttt{fun[x:S]t}]\!]_\rho = [\![\texttt{fun[x:S]t}']\!]_\rho.$$

For instance, we have $[\![0]\!]_{\{x\mapsto v\}} = 0 = [\![\texttt{(fun[y:Nat]0)x}]\!]_{\{x\mapsto v\}}$, but not necessarily $[\![\texttt{fun[x:Nat]0}]\!] = [\![\texttt{fun[x:Nat] (fun[y:Nat]0)x}]\!]$, because the semantical lambda abstraction is defined on descriptions, not on actual functions. This possibility represents a failure of **extensionality** of the semantics.

Another problem with the subset semantics is that, since subtypes are modeled as subsets, the model fails to validate some expected typed equations between records, such as the one we saw above:

$$\texttt{\{x=0,y=0\}} = \texttt{\{x=0,y=1\}} \in \texttt{\{|x:Nat|\}}$$

22.2 The PER interpretation

We can obtain these missing equations by endowing each type interpretation with its own "local" notion of equality. In the case of function types, this equality will be defined in such a way that algorithms that map equal arguments to equal results are regarded as equal. (This will solve both of the problems observed at the end of the previous Section, since records are interpreted as partial functions on labels.)

22.2.1 Definition: A **partial equivalence relation** (PER) is a symmetric and transitive relation (not, in general, reflexive) on the set $\mathbb{N}$ of natural numbers. □

We write mRn when m and n are related by R; alternatively, we can view R as a subset of $\mathbb{N} \times \mathbb{N}$ and write $(m, n) \in R$. When R is a PER, we write *dom*(R) for the set $\{n \mid nRn\}$, the set of numbers related to themselves by R. Notice that R is an equivalence relation on *dom*(R), so a PER can be thought of as consisting of a subset of $\mathbb{N}$ together with a local equality relation. Also, note that if xRy then both x and y are in *dom*(R).

If R and S are PERs, we write $R \subseteq S$ to mean that R is a subrelation of S in the set-theoretic sense—i.e., xRy implies xSy. This means that *dom*(S) is a superset of *dom*(R) and that the equality in S on the common part is coarser than the one in R. This will account for the "non-injective" nature of the subtyping relation between record types (the fact that two different records in a type `S` can become equal when viewed at a supertype `T`).

Interpretation of Types

The types of $F_{<:}$ are interpreted as PERs as follows. (The environment ρ here assigns PERs, not just sets of numbers, to type variables—and, as before, natural numbers to term variables.)

$$
\begin{array}{lcl}
[\![\texttt{Nat}]\!]_\rho & = & \{(n,n) \mid n \in \mathbb{N}\} \\
[\![\texttt{Top}]\!]_\rho & = & \mathbb{N} \times \mathbb{N} \quad \text{(the everywhere-true relation)} \\
[\![\texttt{S}\to\texttt{T}]\!]_\rho & = & \{(t,t') \mid \forall (x,x') \in [\![\texttt{S}]\!]_\rho.\ \{t\}(x)[\![\texttt{T}]\!]_\rho\{t'\}(x')\} \\
[\![\{\!|\texttt{l}_1\texttt{:T}_1 \ldots \texttt{l}_n\texttt{:T}_n|\!\}]\!]_\rho & = & \{(t,t') \mid \forall \texttt{l}_i \in \{\texttt{l}_1 \ldots \texttt{l}_n\}.\ (\{t\}\texttt{l}_i, \{t'\}\texttt{l}_i) \in [\![\texttt{T}_i]\!]_\rho\} \\
[\![\forall\texttt{[X<:S]T}]\!]_\rho & = & \{(t,t') \mid t[\![\texttt{T}]\!]_{\rho+\{\texttt{X}\mapsto U\}}t' \text{ for all } U \subseteq [\![\texttt{S}]\!]_\rho\} \\
 & & \text{that is, } \bigcap_{U \subseteq [\![\texttt{S}]\!]_\rho} [\![\texttt{T}]\!]_{\rho+\{\texttt{X}\mapsto U\}}
\end{array}
$$

22.2.2 Exercise [Quick check]: Show that, if $f, g \in dom([\![\texttt{S}\to\texttt{T}]\!]_\rho)$, then $f[\![\texttt{S}\to\texttt{T}]\!]_\rho g$ iff $f(a)[\![\texttt{T}]\!]_\rho g(a)$ for all $a \in dom([\![\texttt{S}]\!])$. □

The interpretation of terms is exactly as in the subset interpretation in Section 22.1. To show that this interpretation is sound, we must slightly modify the definition of satisfaction between environments and contexts: when `x:S` is a binding in Γ, we require that $\rho(x) \in dom([\![\texttt{S}]\!]_\rho)$.

22.2.3 Theorem [Semantic soundness of the PER intepretation]: The PER interpretation is sound in the sense that all statements are validated semantically. If $\rho \models \Gamma$, then:

1. $\Gamma \vdash \texttt{S} <: \texttt{T}$ implies $[\![\texttt{S}]\!]_\rho \subseteq [\![\texttt{T}]\!]_\rho$;
2. $\Gamma \vdash \texttt{t} : \texttt{S}$ implies $[\![\texttt{t}]\!]_\rho \in dom([\![\texttt{S}]\!]_\rho)$;
3. $\texttt{t} \longrightarrow^* \texttt{t}'$, with $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\Gamma \vdash \texttt{t}' : \texttt{T}$, implies $[\![\texttt{t}]\!]_\rho[\![\texttt{T}]\!]_\rho[\![\texttt{t}']\!]_\rho$. □

Proof: Exercise. The argument is similar to the proof of 22.1.2. First, one establishes the validity of the semantic substitution lemma, replacing subsets by PERs. The proofs then proceed by induction on derivations. In part (2), the statement must be strengthened as follows:

2′. If $\rho \models \Gamma$ and $\rho' \models \Gamma$, and if $\rho(\texttt{X}) = \rho'(\texttt{X})$ for all type variables `X` declared in Γ and $\rho(\texttt{x})[\![\texttt{S}]\!]_\rho\rho'(\texttt{x})$ for all term variable declarations `x:S`, then $\Gamma \vdash \texttt{t} : \texttt{S}$ implies $[\![\texttt{t}]\!]_\rho[\![\texttt{S}]\!]_\rho[\![\texttt{t}]\!]_{\rho'}$.

Notice that $[\![\texttt{S}]\!]_\rho = [\![\texttt{S}]\!]_{\rho'}$ □

One technical note is in order. Strictly speaking, the PER semantics does not model reduction as equality, but rather as relatedness in a certain PER. If one desires actual equality, one can decree that the meaning of a term is not a natural number, but rather an equivalence class of natural numbers; then, however, the meaning of a term depends on its type, which makes the interpretation of subtyping more complicated.

Applications of the PER model

Now we come to the promised application of the PER model as a means to establish observational equivalence of programs. The main idea is that, at the atomic type `Nat`, the semantic equality agrees with the syntactic equality given by the reduction relation.

22.2.4 Lemma: If $\mathtt{t}_1$ and $\mathtt{t}_2$ are closed terms of type `Nat`, then $[\![\mathtt{t}_1]\!][\![\mathtt{Nat}]\!][\![\mathtt{t}_2]\!]$ implies that $\mathtt{t}_1$ and $\mathtt{t}_2$ have the same number as normal form. □

Proof: By the strong normalization of System $F_{<:}$ (Theorem **??**) together with preservation (**??**), $\mathtt{t}_1$ and $\mathtt{t}_2$ each have some number as normal form. By the soundness theorem (22.2.3, part 3), the interpretation of a term is related to the interpretation of its normal form, so the normal forms of $\mathtt{t}_1$ and $\mathtt{t}_2$ are related by $[\![\mathtt{Nat}]\!]$, hence equal. □

The proof of this lemma relies heavily on the strong normalization of $F_{<:}$. An analogous property also holds in the absence of strong normalization (e.g., in the presence of the fixed-point operator). Here, one must show that, if the interpretation of a term is a number, then the term admits a reduction to this number.

22.2.5 Exercise: Give this argument in detail. □

22.2.6 Definition: Two closed terms $\mathtt{t}_1$, $\mathtt{t}_2$ of type `S` are **observationally equivalent at type** `S` (written $\mathtt{t}_1 \overset{\text{obs}}{=}_{\mathtt{S}} \mathtt{t}_2$ or, when `S` is clear from context, just $\mathtt{t}_1 \overset{\text{obs}}{=} \mathtt{t}_2$) if, for every closed term $\mathtt{c} : \mathtt{S}\to\mathtt{Nat}$, it holds that $\mathtt{c}\ \mathtt{t}_1$ and $\mathtt{c}\ \mathtt{t}_2$ reduce to the same number. □

The idea is that $\mathtt{t}_1 \overset{\text{obs}}{=} \mathtt{t}_2$ means that $\mathtt{t}_1$ and $\mathtt{t}_2$ can be replaced by one another in an arbitrary program, since the visible output of a program must be a number.

22.2.7 Remark: This rather simple definition of observational equivalence relies on the fact that there is a distinguished atomic type in which to make observations and on the fact that open terms can be treated as closed terms using functional abstraction (so that we can regard every term containing `t` as a subphrase is equal to an term of the form `c t`, where `t` does not occur in `c`). In more complex languages where other binders are available, it may be necessary to extend the notion of observational equivalence to so-called **contextual equivalence**, where the observing context is a term with a hole somewhere inside it. In particular, such a context may bind free variables in the observed term. □

22.2.8 Theorem [Computational adequacy]: Let $\mathtt{t}_1$ and $\mathtt{t}_2$ be closed terms of a closed type `S`. If $[\![\mathtt{t}_1]\!][\![\mathtt{S}]\!][\![\mathtt{t}_2]\!]$, then $\mathtt{t}_1 \overset{\text{obs}}{=}_{\mathtt{S}} \mathtt{t}_2$. □

Proof: Suppose $\mathtt{t_1}, \mathtt{t_2} : \mathtt{S}$ and $[\![\mathtt{t_1}]\!]\,[\![\mathtt{S}]\!]\,[\![\mathtt{t_2}]\!]$. If $\mathtt{c} : \mathtt{S}\to\mathtt{Nat}$, then $[\![\mathtt{c}]\!] \in dom([\![\mathtt{S}\to\mathtt{Nat}]\!])$. Thus,

$$[\![\mathtt{c\ t_1}]\!] = \{[\![\mathtt{c}]\!]\}([\![\mathtt{t_1}]\!])\ [\![\mathtt{Nat}]\!]\ \{[\![\mathtt{c}]\!]\}([\![\mathtt{t_2}]\!]) = [\![\mathtt{c\ t_2}]\!].$$

Hence, by Lemma 22.2.4, $\mathtt{c\ t_1}$ and $\mathtt{c\ t_2}$ must have the same number as normal form, and $\mathtt{t_1} \stackrel{\text{obs}}{=} \mathtt{t_2}$. □

The computational adequacy theorem shows how the PER model can be employed to reason about observational equivalence. For example, the two records $\mathtt{p} = \mathtt{\{x=0,y=0\}}$ and $\mathtt{q} = \mathtt{\{x=0,y=1\}}$ are observationally equivalent at type $\{\!|\mathtt{x:Nat}|\!\}$. To see this, note that $[\![\mathtt{p}]\!]\,[\![\{\!|\mathtt{x:Nat}|\!\}]\!]\,[\![\mathtt{q}]\!]$, since $\{[\![\mathtt{p}]\!]\}(x) = \{[\![\mathtt{q}]\!]\}(x)$.

22.2.9 Exercise [Recommended]:

1. Show that if $\mathtt{e}$ and $\mathtt{f}$ are closed, well typed terms, then $\mathtt{e} \stackrel{\text{obs}}{=}_{\mathtt{Top}} \mathtt{f}$.

2. Show that whenever $\vdash \mathtt{p} : \mathtt{T}$ where $\mathtt{T} = \{\!|\mathtt{l_1{:}T_1 \ldots l_n{:}T_n}|\!\}$, we have $\mathtt{p} \stackrel{\text{obs}}{=}_{\mathtt{T}} \mathtt{\{l_1{=}p.l_1 \ldots l_n{=}p.l_n\}}$. □

We now investigate observational equivalence at polymorphic types. As we did with the subset semantics, we can use the PER semantics to argue that types such as $[\![\forall\mathtt{[X]}\ \forall\mathtt{[Y]}\ \mathtt{((X\to Y)\to(X\to Y))\to X\to Y}]\!]$ are empty (showing that no fixed-point operator can be defined). Unlike the subset model, though, the PER model can also be used to constrain the elements of *inhabited* universal types, validating some useful principles of uniformity.

22.2.10 Notation: To reduce clutter in the following, we will often abbreviate $\{t\}(n)$ by $t\,n$ or $t(n)$. □

22.2.11 Proposition: Let $\mathtt{S}$ be an arbitrary type and $U = [\![\forall\mathtt{[X<:S]X\to X}]\!]$. If $f \in dom(U)$, then $f\ U\ \underline{\lambda}x.\,x$. □

Proof: We must show that, for every PER $X \subseteq [\![\mathtt{S}]\!]$ and every $x \in dom(X)$, we have $f(x)\ X\ x$. From $f \in dom(U)$, we know that, for each PER $Y \subseteq [\![\mathtt{S}]\!]$ and $x \in dom(Y)$, we have $f(x) \in dom(Y)$. Suppose that $X \subseteq [\![\mathtt{S}]\!]$ and $x \in dom(X)$. Choose $Y \subseteq X \subseteq [\![\mathtt{S}]\!]$ to be the singleton $\{(x,x)\}$. From $x \in dom(Y)$ we infer $f(x) \in dom(Y)$, so $f(x) = x$. Thus, $f(x)\ X\ x$ for every $x \in dom(X)$. □

Notice that this excludes the possibility of "polymorphically updating" fields of records in the PER model; for example, it follows by a similar argument that every function of type $\forall\mathtt{[X<\{x:Nat\}]Nat\to X}$ must discard its `Nat` argument and return its `X` argument unchanged. In particular, such a function cannot have the effect of replacing the `x` field in its first argument by the number given as its second argument. Also, notice that, by computational adequacy, every closed term of this type must be observationally equivalent to such a constant function. Polymorphic update can be supported by extending the syntax of $F_{<:}$, as we shall see in Chapter 29, but such extensions cannot be interpreted in the PER model as it stands; it is, however, possible to construct a more refined PER model, using a different interpretation of subtyping, in which polymorphic update functions do exist [HP95, Pol96, **?**].

Next we examine the interpretations of impredicative encodings of algebraic datatypes, such as the Church booleans and Church numerals introduced in Section 18.4.

22.2.12 Proposition: Let *CBool* be the PER $[\![\forall\texttt{[X]}\ \texttt{X}\to\texttt{X}\to\texttt{X}]\!]$. Let $tt = \underline{\lambda}x.\ \underline{\lambda}y.\ x$ and $ff = \underline{\lambda}x.\ \underline{\lambda}y.\ y$. We have $ff, tt \in dom(CBool)$ and, whenever $f \in dom(CBool)$, either $f\ CBool\ tt$ or $f\ CBool\ ff$. □

Proof: First, recall that $f \in dom(CBool)$ means that $x\ X\ x'$ and $y\ X\ y'$ imply $f\ x\ y\ X\ f\ x'\ y'$, for every PER X and numbers x, x', y, y'. In particular, if $x, y \in dom(X)$, then $f\ x\ y \in dom(X)$.

Now consider the PER $X = \{(0,0),\ (1,1)\}$, with $dom(X) = \{0,1\}$. Since $f\ 0\ 1 \in dom(X)$, we have either $f\ 0\ 1 = 0$ or $f\ 0\ 1 = 1$. Suppose, without loss of generality, that $f\ 0\ 1 = 0$. We claim that, in this case, $f\ CBool\ \texttt{tt}$. Letting x and y be natural numbers, we must show that $f\ x\ y = x$. Using the same argument as before, with 0 and 1 replaced by x and y, we find that $f\ x\ y \in \{x, y\}$. So if $x = y$, then $f\ x\ y = x$ and we are done. Suppose, on the other hand, that x and y are different. There are several cases to consider.

- Suppose x and y are both different from 0 and 1. Let X be the PER with equivalence classes $\{x, 0\}$ and $\{y, 1\}$—i.e.,

  $$X = \{(0,0),\ (0,x),\ (x,0),\ (x,x),\ (1,1),\ (1,y),\ (y,1),\ (y,y)\}.$$

  From $x\ X\ 0$ and $y\ X\ 1$, we get $(f\ x\ y)\ X\ (f\ 0\ 1)$ so $(f\ x\ y)\ X\ 0$ and $f\ x\ y = x$, using the fact that $f\ x\ y$ is either x or y, hence not 0.

- Suppose x is 0. Let X be the PER with equivalence classes $\{0\}$ and $\{y, 1\}$. Since $x \neq y$, these two classes are different. Reasoning as before, we have $(f\ x\ y)\ X\ 0$, so $f\ x\ y = 0 = x$.

- Suppose $x = 1$ and $y = 0$. Let X be the PER with equivalence classes $\{0, 2\}$ and $\{1\}$ and let Y be the PER with equivalence classes $\{0, 1\}$ and $\{2\}$. From $2\ X\ 0$ and $1\ X\ 1$, we obtain $f\ 2\ 1 = 2$. From $2\ Y\ 2$ and $0\ Y\ 1$, we get $f\ 2\ 0 = f\ 2\ 1 = 2$. Finally, using Z with classes $\{1, 2\}$ and $\{0\}$, we get $(f\ 1\ 0) Z (f\ 2\ 0)$, and hence $f\ 1\ 0 \in \{1, 2\}$. Since $f\ x\ y \in \{x, y\}$, we conclude that $f\ 1\ 0 = 1$.

The other cases are left as exercises. □

Similar in spirit but more complex is Freyd's proof that every element of the type $[\![$`∀[X] (X→X)→X→X`$]\!]$ must be related to the interpretation of a Church numeral [Fre89].

The above characterisation of Church booleans would be much easier if we were allowed to use arbitrary relations, not just PERs—if it were the case, for example, that $f \in dom(CBool)$ implied $(f\ x\ y)\ R\ (f\ x'\ y')$ whenever $x\ R\ x'$ and $y\ R\ y'$ for some relation R. We could then directly use the relation defined by $0\ R\ x$ and $1\ R\ y$ and wouldn't need any case distinctions. A model in which such relation based reasoning principles are valid for arbitrary polymorphic types (not only booleans) is called **parametric** [Rey83, **?**]. In a parametric model, very powerful principles for polymorphic encodings of inductive types and abstract datatypes are available (cf. [PA93]). However, it is an open problem whether the PER model is parametric, and the only known parametric models are rather syntactic in flavor.

However, some simple instances of parametric reasoning do go through in the PER model.

22.2.13 Proposition: Let $f \in [\![$`∀[X]X→S`$]\!]$, where `X` is not free in `S`. Then f is constant—that is, for each PER X and element $x \in dom(X)$, we have $(f\ x)\ [\![$`S`$]\!]\ (f\ 0)$. □

Proof: Using the everywhere true relation $[\![$`Top`$]\!]$. (Let x be given. Then $x\ [\![$`Top`$]\!]\ 0$, so $(f\ x)\ [\![$`S`$]\!]\ (f\ 0)$.) □

The above example is a a simple case of **representation independence**. One can view $f \in [\![$`∀[X]X→A`$]\!]$ as a piece of code depending on a module that supplies a type X and a constant of that type. (Think of the encoding of existential types in terms of universal types sketched in Exercise **??**.) If there are no other constants or functions defined on this type, then intuitively it should not be possible to make any use of it. The PER model validates this intuition.

Here is another reasoning principle based on the observation that the PER semantics uses neither type annotations nor type applications or abstractions.

22.2.14 Reasoning Principle [Mitchell]: Any two $F_{<:}$ terms of the same type and with equal erasures are observationally equal.

For example, if

```
f : ∀[X] (X→X) → (Nat→X) → X
V <: Top
U <: V
g : V→U
h : Nat→U,
```

then

$$\texttt{f U (fun[u:U]g(u))(h)} \stackrel{\text{obs}}{=}_{\texttt{V}} \texttt{f V (fun[v:V]g(v))(h)}.$$

□

Proof: Exercise. □

Digression on full abstraction

One may ask whether the reasoning method arising from Theorem 22.2.8 is complete, i.e., whether $\mathtt{t}_1 \stackrel{\text{obs}}{=} \mathtt{t}_2$ implies $[\![\mathtt{s}]\!]=[\![\mathtt{t}]\!]$. Semantic models for which this is the case are called **fully abstract**. It is an open problem (as far as we know) whether the PER model is fully abstract, but it seems unlikely for the following reason.

We might attempt to prove full abstraction by induction on the type `S` of `s` and `t`. If $\mathtt{S} = \mathtt{Nat}$, then it clearly holds; but for the function space $\mathtt{S}\to\mathtt{T}$ we encounter the following problem. Suppose $\vdash \mathtt{f}, \mathtt{g} : \mathtt{S}\to\mathtt{T}$ and $\mathtt{f} \stackrel{\text{obs}}{=} \mathtt{g}$. To prove $[\![\mathtt{f}]\!] = [\![\mathtt{g}]\!]$, we have to show that $[\![\mathtt{f}]\!](\nu)\ [\![\mathtt{T}]\!]\ [\![\mathtt{g}]\!](\nu)$ for all $\nu \in \mathit{dom}([\![\mathtt{S}]\!])$. Using full abstraction inductively on `T`, we get $[\![\mathtt{f}]\!](\nu)\ [\![\mathtt{T}]\!]\ [\![\mathtt{g}]\!](\nu)$ for all *definable* ν, i.e., those of the form $[\![\mathtt{t}]\!]$ for some $\vdash \mathtt{t} : \mathtt{S}$. But even $\mathit{dom}([\![\mathtt{Nat}\to\mathtt{Nat}]\!])$ contains non-definable elements (since it contains all total computable functions, and some of these cannot be defined[4] in $F_{<:}$). It could, however, be that every non-definable element is observationally equivalent to a definable one, in which case we could salvage this argument.

22.3 General Recursion and Recursive Types

Although the PER semantics is based on arbitrary computable functions, only the total ones actually arise as denotations. In order to account for general recursion and recursive types, we need a more generous notion of model.

Continuous Partial Orders

We have already seen that the PER model does not account for general recursion—although the elements of the underlying universe ($\mathbb{N}$) encode partial functions, the interpretations of types include only elements encoding functions that are total on the relevant domain. There does not seem to be an easy way of removing this restriction: if we insist on interpreting types as PERs over $\mathbb{N}$, our model will account only for total functions. To model recursive types (and accordingly, general recursion and higher-order partial functions), we must adopt a richer view of the underlying computations. The required machinery comes from the mathematical area of **domain theory**.

22.3.1 Definition: A **domain** is a partial ordering $(D, \sqsubseteq)$ (that is, a set D with a transitive, reflexive, and anti-symmetric ordering $\sqsubseteq$) with the following properties:

1. If $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ is an increasing chain in D then there exists a least upper bound $\bigsqcup_i x_i \in D$. That is, $x_i \sqsubseteq \bigsqcup_i x_i$ for all i and, if $x_i \sqsubseteq y$ for all i, then $\bigsqcup_i x_i \sqsubseteq y$.

[4]This can be seen from a simple diagonalization argument. Suppose that $(f_x)_{x\in\mathbb{N}}$ is an effective enumeration of all closed $F_{<:}$ terms of type $\mathtt{Nat}\to\mathtt{Nat}$. Then $\underline{\lambda}x.\ f_x(x)+1$ is a computable function that is not definable in $F_{<:}$.

2. D has a least element (written $\perp$ or $\perp_D$ and pronounced "bottom"). □

22.3.2 Exercise [Easy]: Show that the least upper bound of each chain in a domain D is uniquely determined, in the sense that, if $x_i \sqsubseteq z$ for all i and, whenever $x_i \sqsubseteq y$ for all i, we have $z \sqsubseteq y$, then $z = \bigsqcup_i x_i$. □

The structures we call domains are often called "omega-complete partial orders (with bottom)," reserving the word "domain" for structures satisfying further properties such as "algebraicity" or "continuity."

Let D and E be domains. We define the space $[D \to E]$ of **continuous functions** from D to E as the set of monotone functions $f \in D \to E$ that preserve suprema, i.e.,

- $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$, and
- $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$ for every increasing chain $(x_i)_{i \in \mathbb{N}}$.

$D \to E$ can be viewed as a domain itself by ordering its elements pointwise, i.e., $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D$. The bottom element and least upper bounds in this domain are also calculated pointwise. Note that we do not require that the elements of $[D \to E]$ should be strict (i.e., map $\perp_D$ to $\perp_E$).

22.3.3 Proposition: Every continuous function $f \in [D \to D]$ has a fixed point given as the least upper bound of the chain $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \ldots$ □

Proof: Exercise. □

22.3.4 Definition: Let L denote the set of natural numbers considered as record labels. (For technical convenience, we continue using numbers as labels.) We define $Rcd(D)$ as the set of functions $f \in L \to D$, ordered pointwise, plus a distinguished least element $\perp_{Rcd(D)}$ (usually written just $\perp$). Again, this forms a domain.

We will use $Rcd(D)$ to interpret record types. The addition of the $\perp$ element allows us to distinguish a nonterminating computation of record type (such as `fix {||} fun[r:{||}] r`) from a record containing an undefined (nonterminating) field (such as `{l=omega}`). This reflects the fact that the call-by-name evaluation relation does not evaluate within the fields of a record, but waits until a field is projected to evaluate it further. □

22.3.5 Definition: The **flat domain of natural numbers** $\mathbb{N}_\perp$ has as elements the natural numbers plus an extra bottom element $\perp$. The ordering is: $x \sqsubseteq y$ iff $x = \perp$ or $x = y$. □

As before, types will be interpreted as partial equivalence relations. Instead of the natural numbers, though, the underlying universe of these PERs will be the domain that forms the least solution of the following equation:

$$D \quad \cong \quad \mathbb{N}_\perp \; + \; [D \to D] \; + \; Rcd(D)$$

That is, we will use a domain D that "contains" the flat domain of natural numbers, the function space $[D \to D]$, and the record domain $Rcd(D)$. Rather than developing a general theory of such equations and their solutions, we simply assert (without proof) the existence of a suitable domain and list its relevant properties:

22.3.6 Proposition: There exists a domain D with functions

$$
\begin{array}{lcllcl}
\mathbb{N} & \in & [\mathbb{N}_\perp \to D] & \mathbb{N}^\star & \in & [D \to \mathbb{N}_\perp] \\
\mathit{fun} & \in & [[D \to D] \to D] & \mathit{fun}^\star & \in & [D \to [D \to D]] \\
\mathit{rcd} & \in & [\mathit{Rcd}(D) \to D] & \mathit{rcd}^\star & \in & [D \to \mathit{Rcd}(D)]
\end{array}
$$

satisfying the following equations (for $\diamond \in \{\mathbb{N}, \mathit{fun}, \mathit{rcd}\}$)

$$
\begin{array}{l}
\diamond(\diamond^\star(x)) = x \\
\diamond(\perp) = \perp_D \\
\diamond^\star(d) = \begin{cases} e & \text{if } d = \diamond(e) \text{ for some } e \\ \perp & \text{otherwise} \end{cases}
\end{array}
$$

and such that every element $d \in D$ can be expressed as $d = \diamond(e)$ for some e, where $\diamond$ is one of $\{\mathbb{N}, \mathit{fun}, \mathit{rcd}\}$. Furthermore, there is an increasing sequence of projection functions

$$\pi_r \in [D \to D] \qquad \text{for all } r \in \mathbb{N}$$

with the following properties:

1. $\pi_r(d) \sqsubseteq d$
2. $\pi_r(\pi_s(d)) = \pi_{\min(r,s)}(d)$
3. $\pi_0(d) = \perp$
4. $\bigsqcup_r \pi_r(d) = d$
5. $D_r \stackrel{\text{def}}{=} \{d \mid \pi_r(d) = d\}$ is a finite set
6. the projections π_r are compatible with the structure of D in the following sense:
   (a) $\pi_{r+1}(\mathbb{N}(n)) = \begin{cases} \mathbb{N}(n) & \text{if } n \leq r \\ \perp & \text{otherwise} \end{cases}$
   (b) $\pi_{r+1}(\mathit{fun}(f)) = \mathit{fun}(\lambda x.\, \pi_r(f(\pi_r(x))))$.
   (c) $\pi_{r+1}(\mathit{rcd}(f)) = \mathit{rcd}(\lambda l.\ \text{if } l \leq r \text{ then } \pi_r(f(l)) \text{ else } \perp)$ □

Proof: See [Gun92] or any other standard text on domain theory. □

In other words, D has $\mathbb{N}_\perp$, $Rcd(D)$, and—in particular—its own function space $[D \to D]$ as "subdomains," and D is obtained as the "limit" of an increasing sequence of finite subsets—the ranges D_r of the π_r. These approximating subdomains can be explicitly constructed by taking the clauses under (6) as an inductive definition. In particular,

$$
\begin{aligned}
D_0 &= \{\perp\} \\
D_1 &= \{\perp, \mathbb{N}(0), \ \mathit{rcd}(\lambda l.\perp)\} \\
D_2 &= \{\perp, \mathbb{N}(0), \ \mathbb{N}(1), \\
&\qquad \mathit{rcd}(\lambda l.\perp), \ \mathit{rcd}(0 \mapsto \mathbb{N}(0), \ l{>}0 \mapsto \perp), \ \mathit{rcd}(0 \mapsto \mathbb{N}(0), \ 1 \mapsto \mathbb{N}(0), \ l{>}1 \mapsto \perp), \\
&\qquad\quad \mathit{rcd}(0 \mapsto \mathit{rcd}(\lambda l.\perp), \ 1 \mapsto \mathit{rcd}(\lambda l.\perp), \\
&\qquad\quad l{>}1 \mapsto \perp), \text{ and 5 more records} \ldots, \\
&\qquad \mathit{fun}(\lambda x \in D_1.\,\mathbb{N}(0)), \ \mathit{fun}(\lambda x \in D_1.\,x), \text{ and 8 more functions} \ldots \}
\end{aligned}
$$

etc.

22.3.7 Exercise:

1. How many elements does D_3 contain?

2. Convince yourself that D_r can be described as $\{\pi_r(d) \mid d \in D\}$.

3. Describe explicitly the action of the projection π_1 viewed as a function from D_2 to D_1. □

22.3.8 Exercise [Recommended]:

1. Call a domain D **finitary** if every increasing chain in D contains a finite number of elements. Show that, if D is finitary, then the least upper bound of every increasing chain in D is an element of the chain. Does this implication hold in the other direction?

2. Note that $\mathbb{N}_\perp$ and each D_r are (trivially) finitary. Give an example of an increasing chain with infinitely many distinct elements in $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$.

3. Consider an increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ in D. Show that $\pi_r(\bigsqcup_i x_i) = \pi_r(x_j)$ for some j. □

22.3.9 Exercise: Give an explicit definition of a continuous function $\mathit{succ} \in [D \to D]$ with the property that $\mathit{succ}(\mathbb{N}(n)) = \mathbb{N}(n+1)$. □

The CUPER Interpretation

As before, we will interpret types as "subsets of D equipped with local notions of equality"—i.e., PERs over D. However, in order to interpret general recursion and recursive types, some additional conditions will be needed.

1. In order to ensure that the fixed-point functional described in Exercise **??** (which sends f to $\mathit{fix}(f) \stackrel{\text{def}}{=} \bigsqcup_i f^i(\bot)$) will be an element of $\mathit{dom}(\llbracket \forall\texttt{[X]}\,\texttt{(X}\to\texttt{X)}\to\texttt{X} \rrbracket)$, we require that every PER used to interpret types contains the pair $(\bot, \bot)$ and is closed under least upper bounds of increasing chains. (Informally, if R is such a PER and f sends R-related elements to R-related elements, then from $\bot\ R\ \bot$ we see by induction that $f^i\bot\ R\ f^i\bot$ for every i. Hence $\mathit{fix}(f) \in \mathit{dom}(R)$.) This condition is called **completeness**.

2. In order to interpret recursive types, we must further require that the PERs used to interpret types be determined by their restrictions to the finite sets D_r. This condition is called **uniformity**.

22.3.10 Definition: A **complete uniform PER** (CUPER) is a symmetric, transitive relation R on D such that:

1. $\bot\ R\ \bot$.

2. If $(x_i)_{i\in\mathbb{N}}$ and $(y_i)_{i\in\mathbb{N}}$ are two increasing chains such that $x_i\ R\ y_i$ for all i, then $\bigsqcup_i x_i\ R\ \bigsqcup_i y_i$ (completeness).

3. $x\ R\ y$ iff $\pi_r(x)\ R\ \pi_r(y)$ for all r (uniformity). □

22.3.11 Exercise: There is some redundancy in this definition: we can either drop (2) or replace the "iff" in (3) by "implies." Show this. □

22.3.12 Definition:

1. The CUPER *Nat* is defined by $\{(\mathbb{N}(n), \mathbb{N}(n)) \mid n \in \mathbb{N}_\bot\}$.

2. The CUPER *Top* is the total relation $D \times D$.

3. When R and S are CUPERs, the CUPER $R \Rightarrow S$ is defined by $\{(\mathit{fun}(f), \mathit{fun}(g)) \mid$ for all $x, y \in D$, if $x\ R\ y$ then $f(x)\ S\ g(y)\}$.

4. If R_1 through R_n are CUPERs and $l_1 \ldots l_n$ are labels, then the CUPER $\{l_1{:}R_1 \ldots l_n{:}R_n\}$ is defined by $\{(\bot,\bot)\} \cup \{(\mathit{rcd}(f), \mathit{rcd}(g)) \mid f, g \in L \to D$ and $f(l_i)\ R_i\ g(l_i)$ for $i = 1 \ldots n\}$.

5. Let I be any set and $(R_i)_{i\in I}$ be an I-indexed family of CUPERs. The CUPER $\bigcap_i R_i$ is defined as the intersection of all the R_i, i.e., $x\ \bigcap_i R_i\ y$ iff $x\ R_i\ y$ for all $i \in I$.

   (When we use this definition below, the index set I will always be the set of sub-CUPERs of some CUPER.)

6. $\perp$ is the CUPER $\{(\perp,\perp)\}$. □

22.3.13 Exercise [Recommended]: Show that all of these constructions actually define CUPERs. □

These constructions on CUPERs allow us to define a semantics that maps types of $F_{<:}$ (without recursive types) to CUPERs and maps terms to elements of D lying in the domains of their types, using essentially the same definition as in Section 22.2 (the interepretations of both types and terms are with respect to an environment, as before). In this semantics, the type *dom*($[\![\forall\texttt{[X]}\ \texttt{(X}\to\texttt{X)}\to\texttt{X}]\!]$) is nonempty; in particular, it contains the function that sends $f \in [D\to D]$ to its least fixed point. Other elements of this type are finite approximations of the fixed point, i.e., functionals mapping functions f to $f^n(\perp)$ for some n.

It remains to show how to interpret recursive types.

Interpreting Recursive Types

22.3.14 Definition: Let R be a CUPER and $r \in \mathbb{N}$. We write $R\lceil_r$ for the restriction of R to the finite set D_r, i.e., $x\ R\lceil_r\ y$ iff $x\ R\ y$ and $\pi_r(x) = x$ and $\pi_r(y) = y$. □

Notice that $R\lceil_0 = \perp$ and $R\lceil_r$ is a sub-CUPER of R and $R\lceil_{r+1}$, for each CUPER R. Also note that, by the uniformity condition, two CUPERs R and S are equal iff $R\lceil_r = S\lceil_r$ for all $r \in \mathbb{N}$.

22.3.15 Definition: A function F mapping CUPERs to CUPERs is called **contractive** if, for each CUPER R and $r \in \mathbb{N}$ we have $F(R)\lceil_{r+1} = F(R\lceil_r)\lceil_{r+1}$. F is called **nonexpansive** if $F(R)\lceil_{r+1} = F(R\lceil_{r+1})\lceil_{r+1}$. □

Note that "F contractive" implies "F non-expansive," since $F(R\lceil_{r+1})\lceil_{r+1} = F(R\lceil_{r+1})\lceil_{r+2}\lceil_{r+1} = F(R)\lceil_{r+2}|\lceil_{r+1} = F(R)\lceil_{r+1}$.

22.3.16 Exercise [Recommended]: Show that $F(R) = R \Rightarrow \mathbb{N}_\perp$ is contractive. Give an example of a nonexpansive function on CUPERs that is not contractive. □

22.3.17 Definition: Let F be a contractive function mapping CUPERs to CUPERs. The CUPER $\mu(F)$ is defined by $x\ \mu(F)\ y$ iff $\pi_r(x)\ F^r(\perp)\ \pi_r(y)$ for all r. □

22.3.18 Lemma: $\mu(F)\lceil_r = F^r(\perp)\lceil_r$. □

Proof: The direction $\mu(F)\lceil_r \subseteq F^r(\perp)\lceil_r$ is immediate from the definitions. For the other direction, we first establish

$$F^{k+1}(\perp)\lceil_k = F^k(\perp)\lceil_k \quad \text{for all } k \in \mathbb{N} \tag{22.1}$$

by calculating as follows: $F^{k+1}(\bot)\restriction_k = F^k(F(\bot)\restriction_0)\restriction_k = F^k(\bot)\restriction_k$. The penultimate step follows by "commuting" $\restriction_k$ with k applications of F, reducing the index by 1 at each step using contractiveness.

Now assume that $xF^r(\bot)\restriction_r y$, hence, $x, y \in D_r$. From $F^{k+1}(\bot)\restriction_{k+1} \supseteq F^{k+1}(\bot)\restriction_k = F^k(\bot)\restriction_k$ we obtain inductively that $\pi_k(x)F^k(\bot)\restriction_k\pi_k(y)$ for all $k \geq r$. (Note that $\pi_k(x) = \pi_k(\pi_r(x)) = \pi_r(x) = x$.)

On the other hand, $\pi_{r-1}(x)F^r(\bot)\restriction_{r-1}\pi_{r-1}(y)$ by uniformity, hence

$$\pi_{r-1}(x)F^{r-1}(\bot)\restriction_{r-1}\pi_{r-1}(y)$$

by Equation 22.1. Proceeding inductively in this way yields $\pi_k(x)F^k(\bot)\restriction_k\pi_k(y)$ for all $k < r$. □

22.3.19 Proposition: $\mu(F)$ is the unique fixed point of F, that is, $F(\mu(F)) = \mu(F)$ and, if $F(R) = R$ then $R = \mu(F)$. □

Proof: To show that $\mu(F)$ is a fixed point of F, it suffices to show that $\mu(F)\restriction_{r+1} = F(\mu(F))\restriction_{r+1}$ for all $r \in \mathbb{N}$. Now, since F is contractive, $F(\mu(F))\restriction_{r+1} = F(\mu(F)\restriction_r)\restriction_{r+1} = F(F^r(\bot)\restriction_r)\restriction_{r+1} = F^{r+1}(\bot)\restriction_{r+1} = \mu(F)\restriction_{r+1}$ (using Lemma 22.3.18 in the second step and the definition of contractiveness in the third step). On the other hand, if $F(R) = R$, we can show $R\restriction_r = mu(F)\restriction_r$ by induction on r. If $r = 0$, both sides are $\bot$. Otherwise, we have $R\restriction_{r+1} = F(R)\restriction_{r+1} = F(R\restriction_r)\restriction_{r+1} = F(\mu(F)\restriction_r\restriction_{r+1} = \mu(F)\restriction_{r+1}$, where the induction hypothesis is used in the penultimate step. □

22.3.20 Proposition: All functions on CUPERs definable from the operations in Definition 22.3.12 plus μ are nonexpansive; they are contractive if their outermost operation is record formation or function space formation. □

Proof: Straightforward induction (see [AC96]). □

22.3.21 Exercise [Possibly difficult]: We believe that the only nonexpansive but not contractive function that can be constructed using these operators is the identity. Prove or refute this claim. □

It follows that, if a function F on CUPERs is definable from the operations in Definition 22.3.12 plus μ, then the function G mapping each CUPER R to *Top* $\Rightarrow$ $F(R)$ is contractive. This enables us to interpret recursive types by adding the following semantic clauses:

$$\begin{array}{lcl}
[\![\mu(\mathtt{X})\mathtt{T}]\!]_\rho & = & \mu(\lambda R.\,\mathit{Top} => [\![\mathtt{T}]\!]_{\rho+\{\mathtt{X}\mapsto\mathtt{R}\}}) \\
[\![\mathtt{fold}\ (\mu(\mathtt{X})\mathtt{T})]\!]_\rho & = & \mathit{fun}(\lambda d.\,\mathit{fun}(\lambda x.\,d)) \\
[\![\mathtt{unfold}\ (\mu(\mathtt{X})\mathtt{T})]\!]_\rho & = & \mathit{fun}(\lambda d.\,\mathit{fun}^\star(d)(0))
\end{array}$$

For the last two clauses, we use the fact that $[\![\mu(\mathtt{X})\mathtt{T}]\!]_\rho = \mathit{Top} \Rightarrow [\![\{\mathtt{X} \mapsto \mu(\mathtt{X})\mathtt{T}\}\mathtt{T}]\!]_\rho$, which follows from Proposition 22.3.20 and an appropriate substitution lemma.

This semantics shows us (somewhat trivially) that the reduction rules for the system including recursive types do not equate different natural numbers. It also shows the soundness of an equational theory for $F_{<:\mu}$ including additional equations like

$$\texttt{fold } (\mu(\texttt{X})\texttt{T}) \ (\texttt{unfold } (\mu(\texttt{X})\texttt{T}) \ \texttt{t}) = \texttt{t} \in \mu(\texttt{X})\texttt{T}.$$

An appropriate definition of observational equivalence for $F_{<:\mu}$ must deal with the possibility of nontermination—for example, as follows:

> Two closed terms $\texttt{t}_1$ and $\texttt{t}_2$ of type `T` are observationally equivalent (at type `T`) if, for all closed terms $\texttt{C} : \texttt{T} \rightarrow \texttt{Nat}$, the terms $\texttt{C t}_1$ and $\texttt{C t}_2$ either both diverge (i.e., admit no normal form) or can be reduced to the same number.

Computational adequacy of the CUPER semantics with respect to this notion of equivalence now becomes a nontrivial result. We must show that, if `m` is a closed term of type `Nat`, then $[\![\texttt{m}]\!] = \perp$ iff `m` has no normal form. One direction is trivial (which?). The other requires a "logical relations" proof, i.e., an induction over terms using a suitable generalization of the desired property for types other than `Nat`. (We are not aware of such a proof in the literature, but there seem to be no major obstacles.)

22.4 Further Reading

The subset and PER models of the simply typed lambda-calculus are due to Kreisel, who called them HRO ("hereditarily recursive operations") and HEO ("hereditarily effective operations"), respectively. A good reference with pointers to related topics of historical interest is Troelstra and van Dalen's book [TvD88]. A standard reference on PER models in a computer science context is Mitchell's book [Mit96].

Our presentation of CUPERs is a simplified version of the one used by Abadi and Cardelli [AC96]. Other treatments of CUPERs can be found in [**?**, **?**]

The concept of full abstraction was introduced by Plotkin in [Plo77], which is still well worth reading. Recent progress in the construction of fully abstract models (albeit not yet for $F_{<:}$) is reported in [AJM94, HO94, OR94].

Parametricity was first studied by Reynolds in the context of representation independence [Rey83]. Parametric models of System F are described in [BFSS90, RR94]. Parametricity in System $F_{<:}$ is considered in [CMMS94].

Good general textbooks on semantics include [Sch86, Gun92, Ten81, Win93, Mit96].

Chapter 23

Type Equivalence

To be written.

The point of introducing equivalence as a separate notion is that it gives a uniform framework for all kinds of definitional equivalences on types. It introduces a bit of overhead too, but I think it's worth it in this case.

This chapter has just been added, and some of the later chapters have not been brought up to date (the equivalence rules used to be folded into the subtyping relation).

Records with permutation of fields $\rightarrow$ {} $\leftrightarrow$

Type equivalence $\boxed{S \leftrightarrow T}$

$$T \leftrightarrow T \qquad \text{(Q-REFL)}$$

$$\frac{T \leftrightarrow S}{S \leftrightarrow T} \qquad \text{(Q-SYMM)}$$

$$\frac{S \leftrightarrow U \qquad U \leftrightarrow T}{S \leftrightarrow T} \qquad \text{(Q-TRANS)}$$

$$\frac{S_1 \leftrightarrow T_1 \qquad S_2 \leftrightarrow T_2}{S_1 \rightarrow S_2 \leftrightarrow T_1 \rightarrow T_2} \qquad \text{(Q-ARROW)}$$

$$\frac{\pi \text{ is a permutation of } \{1..n\}}{\{l_i : T_i \;^{i \in 1..n}\} \leftrightarrow \{l_{\pi i} : T_{\pi i} \;^{i \in 1..n}\}} \qquad \text{(Q-RCD-PERM)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t : S \qquad S \leftrightarrow T}{\Gamma \vdash t : T} \qquad \text{(T-EQ)}$$

Algorithmic rules for simply typed lambda-calculus $\rightarrow$ B

Algorithmic typing $\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \quad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(TA-ABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = T_{11} \rightarrow T_{12} \qquad \Gamma \mapsto t_2 : T_2 \qquad T_2 = T_{11}}{\Gamma \mapsto t_1\ t_2 : T_{12}} \quad \text{(TA-APP)}$$

Algorithmic rules for records with permutation $\rightarrow$ {} $\leftrightarrow$

Algorithmic type equivalence $\boxed{\mapsto S \leftrightarrow T}$

$$\frac{\mapsto S_1 \leftrightarrow T_1 \qquad \mapsto S_2 \leftrightarrow T_2}{\mapsto S_1 \rightarrow S_2 \leftrightarrow T_1 \rightarrow T_2} \quad \text{(QA-ARROW)}$$

$$\frac{\begin{array}{c}\pi \text{ is a permutation of } \{1..n\} \\ \text{for each } i \quad \mapsto T_i \leftrightarrow T_{\pi i}\end{array}}{\mapsto \{l_i{:}T_i{}^{i\in 1..n}\} \leftrightarrow \{l_{\pi i}{:}T_{\pi i}^{i\in 1..n}\}} \quad \text{(QA-RCD)}$$

Algorithmic typing $\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \quad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(TA-ABS)}$$

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad T_1 = T_{11} \rightarrow T_{12} \qquad \Gamma \mapsto t_2 : T_2 \qquad \mapsto T_2 \leftrightarrow T_{11}}{\Gamma \mapsto t_1\ t_2 : T_{12}} \quad \text{(TA-APP)}$$

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

$$\frac{\Gamma \mapsto t : \{l_1{:}T_1 \ldots l_n{:}T_n\}}{\Gamma \mapsto t.l_i : T_i} \quad \text{(TA-PROJ)}$$

Chapter 24

Definitions

24.1 Type Definitions

(needed for equirec implementation stuff)

introduce type variables (this section is optional, so we need to re-introduce them when we get to polymporphism)

extend type equivalence relation

talk about exposing types before matching against them

Exercise: inner defns (not trivial: must eliminate remaining occurrences when leaving the scope of the definition)

Type definitions $\rightarrow$ B $\leftrightarrow$

New syntactic forms

T	::=	...	*types*
		X	*type variable*
Γ	::=	...	*contexts*
		Γ, X↔T	*type definition binding*

New type equivalence rules $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\Gamma \vdash T \leftrightarrow T \tag{Q-Refl}$$

$$\frac{\Gamma \vdash T \leftrightarrow S}{\Gamma \vdash S \leftrightarrow T} \tag{Q-Symm}$$

$$\frac{\Gamma \vdash S \leftrightarrow U \qquad \Gamma \vdash U \leftrightarrow T}{\Gamma \vdash S \leftrightarrow T} \tag{Q-Trans}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 \qquad \Gamma \vdash S_2 \leftrightarrow T_2}{\Gamma \vdash S_1 \to S_2 \leftrightarrow T_1 \to T_2} \qquad \text{(Q-ARROW)}$$

$$\frac{X \leftrightarrow T \in \Gamma}{\Gamma \vdash X \leftrightarrow T} \qquad \text{(Q-DEF)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S \leftrightarrow T}{\Gamma \vdash t : T} \qquad \text{(T-EQ)}$$

Algorithmic rules for type definitions $\to$ B $\leftrightarrow$

Algorithmic reduction $\boxed{\Gamma \mapsto T \rightarrowtail T'}$

$$\frac{X \leftrightarrow T \in \Gamma}{\Gamma \vdash X \rightarrowtail T} \qquad \text{(RA-DEF)}$$

Algorithmic type equivalence $\boxed{\Gamma \mapsto S \leftrightarrow T}$

$$\Gamma \mapsto B \leftrightarrow B \qquad \text{(QA-BASE)}$$

$$\frac{\Gamma \mapsto S \rightarrowtail S' \qquad \Gamma \mapsto S' \leftrightarrow T}{\Gamma \mapsto S \leftrightarrow T} \qquad \text{(QA-REDUCEL)}$$

$$\frac{\Gamma \mapsto T \rightarrowtail T' \qquad \Gamma \mapsto S \leftrightarrow T'}{\Gamma \mapsto S \leftrightarrow T} \qquad \text{(QA-REDUCER)}$$

$$\frac{\Gamma \mapsto S_1 \leftrightarrow T_1 \qquad \Gamma \mapsto S_2 \leftrightarrow T_2}{\Gamma \mapsto S_1 \to S_2 \leftrightarrow T_1 \to T_2} \qquad \text{(QA-ARROW)}$$

Exposure $\boxed{\Gamma \mapsto T \Uparrow T'}$

$$\frac{\text{otherwise}}{\Gamma \mapsto T \Uparrow T} \qquad \text{(XA-OTHER)}$$

$$\frac{\Gamma \mapsto T \rightarrowtail T' \qquad \Gamma \mapsto T' \Uparrow T''}{\Gamma \mapsto T \Uparrow T''} \qquad \text{(XA-REDUCE)}$$

Algorithmic typing $\boxed{\Gamma \mapsto t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \mapsto x : T} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \mapsto t_2 : T_2}{\Gamma \mapsto \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_1 \Uparrow (\mathtt{T}_{11} \to \mathtt{T}_{12}) \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_2 \leftrightarrow \mathtt{T}_{11}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(TA-APP)}$$

24.2 Term Definitions

(This should be very optional, but it's nice to be honest about what the checkers are doing.)

Chapter 25

Type Operators and Kinding

The introduction to this chapter is in good shape. The technicalities are all missing at the moment.

In previous chapters, we have often made use of informal abbreviations like

```
Pair S T = ∀X. (S→T→X) → X;
```

for brevity and readability, writing `Pair Nat Bool`, for example, instead of the more cumbersome `∀X. (Nat→Bool→X) → X`. What sort of thing is `Pair` in itself? Given any pair of types `S` and `T`, it picks out the type of "pairs of `S` and `T`"—that is, it is intuitively a *function* from types to types. The definition of `Pair` can thus be written explicitly as follows:

```
Pair = λA. λB. ∀X. (A→B→X) → X;
```

Formally, then, the type expression `Pair S T` is actually a nested operator application `(Pair S) T`. In other words, `Pair` itself is a function from types to type operators mapping types to types. In this chapter, we take a more careful look at the mechanisms involved in defining and using such functions.

25.1 Intuitions

Once we have introduced syntax for abstracting type expressions on type expressions (λ) and performing the corresponding instantiations (application), nothing prevents us from considering functions mapping type operators to types, functions mapping type operators to type operators, etc. To keep things organized and prevent ourselves from writing nonsensical type applications like `Pair Pair`, we classify types and type operators by **kinds**. E.g.,

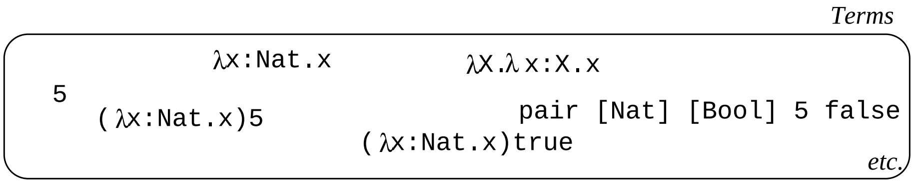
$*$	the kind of "proper types" like `Bool` and `Bool→Bool`
$*\to*$	the kind of type operators (functions from types to types)
$*\to*\to*$	the kind of functions from types to type operators
$(*\to*)\to*$	the kind of functions from type operators to types
etc.	

Kinds can be thought of as the "types of types." In essence, the system of kinds is a little copy of the simply typed lambda-calculus, placed one level higher in the hierarchy of classification. Just as in the lambda-calculus, we shall henceforth annotate the bound variables in operator abstractions with the kind of the bound type variable, so that `Pair` is really defined as:

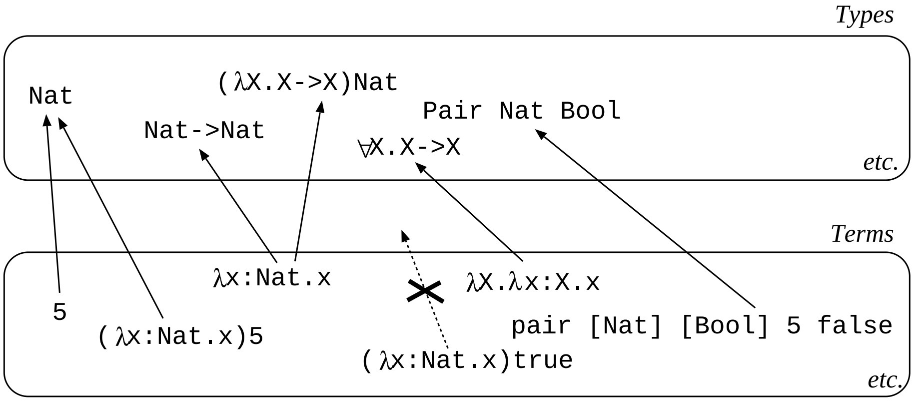
```
Pair = λA::*. λB::*. ∀X. (A→B→X) → X;
```

However, since $*$ is by far the most common case, we allow `λX.T` as an abbreviation for `λX::*.T`.

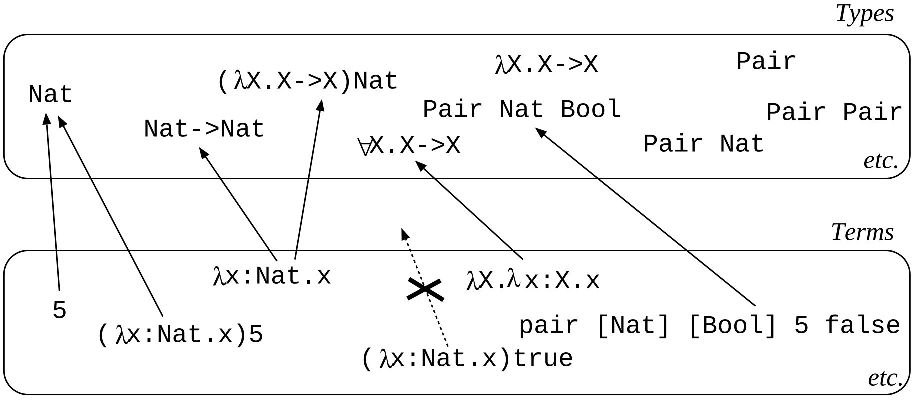
A few pictures should make all this easier to understand. The phrases of our language are divided now into three separate classes: terms, types, and kinds. The level of **terms** contains actual data values (integers, records, etc.) and computations (functions) over them.



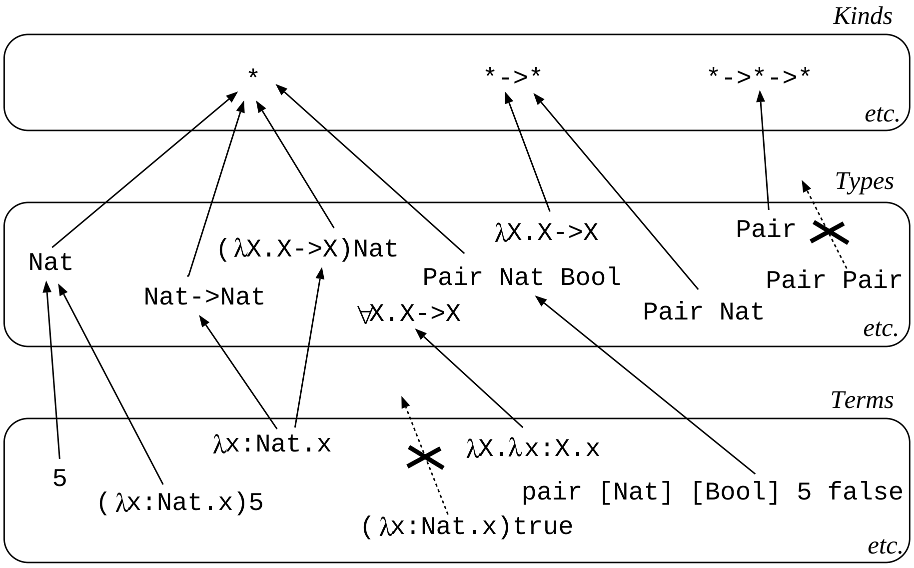
The level of **types** contains **proper types** like `Nat`, `Nat→Nat`, `Pair Nat Bool` and `∀X.X→X`, which classify terms,



as well as **type operators** like `Pair` and λ`X.X`$\to$`X` that do not themselves classify terms but that can be applied to type arguments to form type expressions like (λ`X.X`$\to$`X)Nat` that do classify terms. (We use the word "types" to include both proper types and type operators.)



Proper types are classified by the kind $*$, pronounced "kind type" or just "type." Type operators are classified by more complex kinds, such as $*\Rightarrow*$ and $*\Rightarrow*\Rightarrow*$.



Note that proper types (type expressions of kind $*$) can include type operators of higher kinds as subphrases, as in (λ`X`::$*$`.X`$\to$`X)Nat` or `Pair Nat Bool`.

Also, note the different roles in this diagram of the proper type $\forall$`X.X`$\to$`X` and the type operator λ`X.X`$\to$`X`. The former classifies terms like λ`X.`λ`x:X.x`—it is a type whose *elements* are functions from types to terms—while the latter is *itself* a function from types to types, and does not have any terms as "elements."

A natural question to ask at this point is "Why stop at three levels of expressions?" That is, why couldn't we go on to introduce functions from kinds to kinds, application at the level of kinds, etc., and add a fourth level to classify kind expressions according to their functionality? In fact, why stop there? We could go on adding levels indefinitely! The answer is that we could very well do this, and such systems (without subtyping) have been studied under the heading of **pure type systems** [Bar92a, Bar92b, JMP94, MP93, Pol94, etc.] and used in computer science for applications such as theorem proving. For purposes of this book, though, there is no need to go beyond three levels.

25.2 Definitions

We now define a core calculus with type operators. At the term level, it includes just the variables, abstraction, and application of the simply typed lambda-calculus; the type level includes the usual arrow types and type variables, plus operator abstraction and application. To formalize this system, we need to add three new things to the systems we have seen before.

First, we need a collection of rules of **kinding** (or, more pedantically, **type well-formedness**), which specify how type expressions can be combined to yield new type expressions. In effect, these rules constitute a copy of the simply typed lambda-calculus, "one level up." We write $\Gamma \vdash \mathtt{T} :: \mathtt{K}$ for "type `T` has kind `K` in context Γ."

Second, whenever a type `T` appears in a term (as in λ`x:T.t`), we must check that `T` is well formed. This involves adding a new premise to the old T-ABS rule that checks $\Gamma \vdash \mathtt{T} :: *$. Note that `T` must have exactly kind $*$—i.e., it must be a proper type—since it is being used to describe the values that `x` may range over.

Third, note that, by introducing abstraction and application in types, we have given ourselves the ability to write different "names" for the same type. For example, if `Id` $= \lambda$`X.X`, then

```
Nat → Bool
Nat → Id Bool
Id Nat → Id Bool
Id Nat → Bool
Id (Id (Id Nat)) → Bool
```

all mean the same thing, in the sense that they have the same terms as members.

Type operators and kinding $\rightarrow \leftrightarrow \Rightarrow$

Syntax

$t ::=$		*terms*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
$v ::=$		*values*
	$\lambda x{:}T.t$	*abstraction value*
$T ::=$		*types*
	X	*type variable*
	$\lambda X{::}K.T$	*operator abstraction*
	$T\ T$	*operator application*
	$T \rightarrow T$	*type of functions*
$\Gamma ::=$		*contexts*
	$\emptyset$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*
	$\Gamma, X{::}K$	*type variable binding*
$K ::=$		*kinds*
	$*$	*kind of proper types*
	$K \Rightarrow K$	*kind of operators*

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

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

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

Type equivalence $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \leftrightarrow T :: K} \qquad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash T \leftrightarrow S :: K}{\Gamma \vdash S \leftrightarrow T :: K} \qquad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash S \leftrightarrow U :: K \qquad \Gamma \vdash U \leftrightarrow T :: K}{\Gamma \vdash S \leftrightarrow T :: K} \qquad \text{(Q-TRANS)}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_{11} \vdash \mathtt{T}_{12} :: \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash (\lambda\mathtt{X}::\mathtt{K}_{11}.\mathtt{T}_{12})\ \mathtt{T}_2 \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12} :: \mathtt{K}_{12}} \quad \text{(Q-BETA)}$$

$$\frac{\Gamma \vdash \mathtt{S} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{T} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}\ \mathtt{X} \leftrightarrow \mathtt{T}\ \mathtt{X} :: \mathtt{K}_2}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} : \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \quad \text{(Q-EXT)}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda\mathtt{X}::\mathtt{K}_1.\mathtt{S}_2 \leftrightarrow \lambda\mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \quad \text{(Q-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{S}_1\ \mathtt{S}_2 \leftrightarrow \mathtt{T}_1\ \mathtt{S}_2 :: \mathtt{K}_{12}} \quad \text{(Q-APP)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 \leftrightarrow \mathtt{T}_1 \to \mathtt{T}_2 :: *} \quad \text{(Q-ARROW)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X}::\mathtt{K} \in \Gamma}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \quad \text{(K-TVAR)}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda\mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \quad \text{(K-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{T}_1\ \mathtt{T}_2 :: \mathtt{K}_{12}} \quad \text{(K-TAPP)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{T}_1 \to \mathtt{T}_2 :: *} \quad \text{(K-ARROW)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad \text{(T-EQ)}$$

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

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

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

Abbreviations

unannotated binder for $\mathtt{X} \stackrel{\text{def}}{=} \mathtt{X}::*$

Chapter 26

Higher-Order Polymorphism

Just a sketch.

I'm not certain whether it's productive to develop metatheory in detail for this part of the book, or whether to refer interested readers into the literature. For F-omega itself, the proofs are not all that hard or tedious, so it may be illuminating to go through at least some of them. For later systems like F-omega-sub, it becomes increasingly heavy. (One purpose of putting some of that in would probably be to give students the impression that it's not that easy, in case they thought type systems was a trivial topic!)

When type operators are added to System F, we can abstract over types of arbitrary kinds. This substantially increases the power of the system. In particular, higher-order universal polymorphism (i.e., abstraction over types of kinds other than $*$) will be used in Chapter 30 to complete our model of purely functional object-oriented programming. Higher-order existential types have more practical uses, for building abstract data types whose hidden representation type is actually an operator like `List` or `Pair`.

26.1 Higher-Order Universal Types

The higher-order extension of the polymorphic lambda-calculus is formed by simply allowing `X::K` in place of just `X` in the original definition of System F.

For example, here is a ridiculously polymorphic identity function

```
wayPolyId = λF::*⇒*. λX. λx:F X. x;
▸ wayPolyId : ∀F::*⇒*. ∀X. F X → F X
```

and here is a typical use:

```
wayPolyId [λY.{a:Y}] [Bool] {a=true};
```

26.1.1 Exercise: Write and typecheck a ridiculously polymorphic `applyTwice` function, of type $\forall$F::$*$⇒$*$. $\forall$X. ((F X) → (F X)) → (F X). □

We will see some more interesting applications of higher-order universal types in later chapters.

F^{ω} : Higher-order polymorphic lambda-calculus **System F + ⇒**

New syntactic forms

t ::=	...	*terms*
	λX::K.t	*type abstraction*
v ::=	...	*values*
	λX::K.t	*type abstraction value*
T ::=	...	*types*
	$\forall$X::K.T	*universal type*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$(\lambda X{::}K_{11}.t_{12})\ [T_2] \longrightarrow \{X \mapsto T_2\}t_{12} \qquad \text{(E-BETA2)}$$

New type equivalence rules $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\frac{\Gamma, X{::}K_1 \vdash S_2 \leftrightarrow T_2 :: *}{\Gamma \vdash \forall X{::}K_1.S_2 \leftrightarrow \forall X{::}K_1.T_2 :: *} \qquad \text{(Q-ALL)}$$

New kinding rules $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma, X{::}K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{::}K_1.T_2 :: *} \qquad \text{(K-ALL)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma, X{::}K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{::}K_1.t_2 : \forall X{::}K_1.T_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{::}K_{11}.T_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}T_{12}} \qquad \text{(T-TAPP)}$$

26.2 Higher-Order Existential Types

Similarly, the higher-order variant of existential types is obtained by generalizing `X` to `X::K` in the original definition of λ∃.

Higher-order existential types $F^{\omega}_{<:}$ + ∃

New syntactic forms

```
T ::= ...                    types
      {∃X::K,T}              existential type
```

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : \{X \mapsto U\}T_2 \qquad \Gamma \vdash \{\exists X{::}K_1, T_2\} : *}{\Gamma \vdash \{\exists X{=}U, t_2\}\ \text{as}\ \{\exists X{::}K_1, T_2\} : \{\exists X{::}K_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \exists X{::}K_{11}.T_{12} \qquad \Gamma, X{::}K_{11}, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let}\ \{X,x\}{=}t_1\ \text{in}\ t_2 : T_2} \quad \text{(T-UNPACK)}$$

For example, here is an ADT (cf. Section **??**) that provides the type operator `Pair` abstractly:

```
let pairADT =
   {∃Pair = λX. λY. ∀R. (X→Y→R) → R,
    {pair = λX. λY. λx:X. λy:Y.
              λR. λp:X→Y→R. p 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)}}
 as {∃Pair::*⇒*⇒*,
      {pair: ∀X. ∀Y. X→Y→(Pair X Y),
       fst: ∀X. ∀Y. (Pair X Y)→X,
       snd: ∀X. ∀Y. (Pair X Y)→Y
       }}
in let {Pair,pair}=pairADT
in
  pair.fst [Nat] [Bool] (pair.pair [Nat] [Bool] 5 true);
```

26.2.1 Exercise [Recommended]: Write a program that defines the `pair` ADT, opens it, and then defines a `List` ADT with the representation type

```
List = λX. ∀R. (X→R→R) → R → R;
```

and with operations `nil`, `cons`, `car`, and `cdr` of appropriate types. □

F$^\omega$: Summary **System F+ ⇒**

Syntax

`t ::=`		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`λX::K.t`	*type abstraction*
	`t [T]`	*type application*
`v ::=`		*values*
	`λx:T.t`	*abstraction value*
	`λX::K.t`	*type abstraction value*
`T ::=`		*types*
	`X`	*type variable*
	`T→T`	*type of functions*
	`∀X::K.T`	*universal type*
	`λX::K.T`	*operator abstraction*
	`T T`	*operator application*
`K ::=`		*kinds*
	`*`	*kind of proper types*
	`K⇒K`	*kind of operators*
`Γ ::=`		*contexts*
	`∅`	*empty context*
	`Γ, x:T`	*term variable binding*
	`Γ, X::K`	*type variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \tag{E-Beta}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \tag{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \tag{E-App2}$$

$$(\lambda X{::}K_{11}.t_{12})\ [T_2] \longrightarrow \{X \mapsto T_2\}t_{12} \tag{E-Beta2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \tag{E-TApp}$$

Type equivalence $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T} :: \mathtt{K}} \tag{Q-Refl}$$

$$\frac{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{S} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \tag{Q-Symm}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{U} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \leftrightarrow \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \tag{Q-Trans}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \forall \mathtt{X}::\mathtt{K}_1.\mathtt{S}_2 \leftrightarrow \forall \mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: *} \tag{Q-All}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_{11} \vdash \mathtt{T}_{12} :: \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash (\lambda \mathtt{X}::\mathtt{K}_{11}.\mathtt{T}_{12})\ \mathtt{T}_2 \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12} :: \mathtt{K}_{12}} \tag{Q-Beta}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{S} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{T} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \\ \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}\ \mathtt{X} \leftrightarrow \mathtt{T}\ \mathtt{X} :: \mathtt{K}_2\end{array}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} : \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \tag{Q-Ext}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda \mathtt{X}::\mathtt{K}_1.\mathtt{S}_2 \leftrightarrow \lambda \mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \tag{Q-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{S}_1\ \mathtt{S}_2 \leftrightarrow \mathtt{T}_1\ \mathtt{S}_2 :: \mathtt{K}_{12}} \tag{Q-App}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{S}_1 \rightarrow \mathtt{S}_2 \leftrightarrow \mathtt{T}_1 \rightarrow \mathtt{T}_2 :: *} \tag{Q-Arrow}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X}::\mathtt{K} \in \Gamma}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \tag{K-TVar}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \forall \mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: *} \tag{K-All}$$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda \mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \tag{K-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{T}_1\ \mathtt{T}_2 :: \mathtt{K}_{12}} \tag{K-TApp}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{T}_1 \rightarrow \mathtt{T}_2 :: *} \tag{K-Arrow}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \tag{T-Var}$$

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

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

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}::\mathtt{K}_1.\mathtt{t}_2 : \forall \mathtt{X}::\mathtt{K}_1.\mathtt{T}_2} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}::\mathtt{K}_{11}.\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}} \qquad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-EQ)}$$

26.3 Type Equivalence and Reduction

26.3.1 Definition: The type reduction relation $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{T} :: \mathtt{K}$ is defined just like $\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}$, replacing $\leftrightarrow$ by $\Rrightarrow$ and dropping the rule Q-SYMM. □

26.3.2 Proposition: If $\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}$, then there is some $\mathtt{U}$ such that $\Gamma \vdash \mathtt{S} \Rrightarrow \mathtt{U} :: \mathtt{K}$ and $\Gamma \vdash \mathtt{T} \Rrightarrow \mathtt{U} :: \mathtt{K}$. □

Proof: Standard. □

26.4 Soundness

26.4.1 Lemma:

1. If $\Gamma, \mathtt{x}{:}\mathtt{S}, \Delta \vdash \mathtt{T} :: \mathtt{K}$, then $\Gamma, \Delta \vdash \mathtt{T} :: \mathtt{K}$.
2. If $\Gamma, \mathtt{x}{:}\mathtt{S}, \Delta \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}$, then $\Gamma, \Delta \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}$. □

Proof: The kinding and type equivalence relations do not depend on term variable bindings. □

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

Proof: Straightforward. (Quick check: where is Lemma 26.4.1 used?) □

26.4.3 Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\vdash \mathtt{t}' : \mathtt{T}$. □

Proof: Straightforward induction on evaluation derivations, using Lemma 26.4.2 for the T-VAR case. □

26.4.4 Lemma:

1. If $\mathtt{t}$ is a value and $\vdash \mathtt{t} : \mathtt{T_1}{\to}\mathtt{T_2}$, then $\mathtt{t}$ is an abstraction.
2. If $\mathtt{t}$ is a value and $\vdash \mathtt{t} : \forall\mathtt{X}.\mathtt{T_2}$, then $\mathtt{t}$ is a type abstraction. □

Proof: The arguments for the two parts are similar; we show just (1). Since there are only two forms of values, if $\mathtt{t}$ is a value and not an abstraction, then it must be a type abstraction. Suppose (for a contradiction) that it is a type abstraction. Then the given typing derivation for $\Gamma \vdash \mathtt{t} : \mathtt{T_1}{\to}\mathtt{T_2}$ must end with a use of T-TABS followed by at least one use of T-EQ. That is, it must have the following form:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathtt{t} : \forall\mathtt{X}.\mathtt{S_{12}}}\,(\text{T-TABS}) \qquad \Gamma \vdash \forall\mathtt{X}.\mathtt{S_{12}} \leftrightarrow \mathtt{U_1} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{U_1}}\,(\text{T-EQ})$$

$$\vdots$$

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{t} : \mathtt{U}_{n-1} \qquad \Gamma \vdash \mathtt{U}_{n-1} \leftrightarrow \mathtt{U}_n :: *}{\Gamma \vdash \mathtt{t} : \mathtt{U}_n}\,(\text{T-EQ}) \qquad \Gamma \vdash \mathtt{U}_n \leftrightarrow \mathtt{T_1}{\to}\mathtt{T_2} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T_1}{\to}\mathtt{T_2}}\,(\text{T-EQ})$$

Since type equivalence is transitive, we can collapse all of these uses of equivalence into one and conclude that $\Gamma \vdash \forall\mathtt{X}.\mathtt{S_{12}} \leftrightarrow \mathtt{T_1}{\to}\mathtt{T_2}$.

By Proposition 26.3.2, there must be some type $\mathtt{U}$ such that $\Gamma \vdash \forall\mathtt{X}.\mathtt{S_{12}} \Rrightarrow \mathtt{U}$ and $\Gamma \vdash \mathtt{T_1}{\to}\mathtt{T_2} \Rrightarrow \mathtt{U}$. A quick inspection of the rules defining the $\Rrightarrow$ relation shows that there can be no such $\mathtt{U}$. □

26.4.5 Theorem [Progress]: Suppose $\mathtt{t}$ is closed and stuck, and that $\vdash \mathtt{t} : \mathtt{T}$. Then $\mathtt{t}$ is a value. □

Proof: By induction on typing derivations. The T-VAR case cannot occur, because $\mathtt{t}$ is closed. The T-ABS and T-TABS cases are immediate, since abstractions are values. The T-EQ case follows directly from the induction hypothesis. The remaining cases, for application and type application, are more interesting.

Case T-APP: $\quad \mathtt{t} = \mathtt{t_1}\ \mathtt{t_2} \qquad \vdash \mathtt{t_1} : \mathtt{T_{11}}{\to}\mathtt{T_{12}} \qquad \vdash \mathtt{t_2} : \mathtt{T_{11}}$

This case cannot occur. To see this, we reason by contradiction. Looking at the evaluation rules, it is clear that a term of the form $\mathtt{t_1}\ \mathtt{t_2}$ can be stuck only if

1. $\mathtt{t_1}$ is stuck (otherwise E-APP1 would apply to $\mathtt{t}$), and
2. either $\mathtt{t_1}$ is not a value or else $\mathtt{t_2}$ is stuck (otherwise E-APP2 would apply).

From (1) and the induction hypothesis, we see that $\mathtt{t_1}$ must be a value; so by (2) and the induction hypothesis, $\mathtt{t_2}$ must also be a value. But by Lemma 26.4.4(1), $\mathtt{t_1}$ must be an abstraction, which means that E-BETA applies to $\mathtt{t}$, contradicting our assumption that it is stuck.

Case T-TAPP*:*
Similar. □

Chapter 27

Implementing Higher-Order Systems

Algorithmic rules for F^{ω}

Algorithmic reduction $\boxed{\Gamma \Vdash T \rightarrowtail T'}$

$$\Gamma \Vdash (\lambda X{::}K_{11}.\ T_{12})\ T_2 \rightarrowtail \{X \mapsto T_2\}T_{12} \qquad \text{(RA-Beta)}$$

$$\frac{\Gamma \Vdash T_1 \rightarrowtail T_1' \qquad \Gamma \Vdash T_1'\ T_2 \rightarrowtail T'}{\Gamma \Vdash T_1\ T_2 \rightarrowtail T'} \qquad \text{(RA-App)}$$

Algorithmic type equivalence $\boxed{\Gamma \Vdash S \leftrightarrow T}$

$$\Gamma \Vdash X \leftrightarrow X \qquad \text{(QA-TVar)}$$

$$\frac{\Gamma \Vdash S \rightarrowtail S' \qquad \Gamma \Vdash S' \leftrightarrow T}{\Gamma \Vdash S \leftrightarrow T} \qquad \text{(QA-ReduceL)}$$

$$\frac{\Gamma \Vdash T \rightarrowtail T' \qquad \Gamma \Vdash S \leftrightarrow T'}{\Gamma \Vdash S \leftrightarrow T} \qquad \text{(QA-ReduceR)}$$

$$\frac{\Gamma \Vdash S_1 \leftrightarrow T_1 \qquad \Gamma \Vdash S_2 \leftrightarrow T_2}{\Gamma \Vdash S_1 \to S_2 \leftrightarrow T_1 \to T_2} \qquad \text{(QA-Arrow)}$$

$$\frac{\Gamma, X{::}K_1 \Vdash S_2 \leftrightarrow T_2}{\Gamma \Vdash \forall X{::}K_1.S_2 \leftrightarrow \forall X{::}K_1.T_2} \qquad \text{(QA-All)}$$

$$\frac{\Gamma, X{::}K_1 \Vdash S_2 \leftrightarrow T_2}{\Gamma \Vdash \lambda X{::}K_1.S_2 \leftrightarrow \lambda X{::}K_1.T_2} \qquad \text{(QA-TAbs)}$$

$$\frac{\Gamma \Vdash S_1 \leftrightarrow T_1 \qquad \Gamma \Vdash S_2 \leftrightarrow T_2}{\Gamma \Vdash S_1\ S_2 \leftrightarrow T_1\ T_2} \qquad \text{(QA-TApp)}$$

Exposure $\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T} \Uparrow \mathtt{T}'}$

$$\frac{\text{otherwise}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T} \Uparrow \mathtt{T}} \qquad \text{(XA-OTHER)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T} \rightarrowtail \mathtt{T}' \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}' \Uparrow \mathtt{T}''}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T} \Uparrow \mathtt{T}''} \qquad \text{(XA-REDUCE)}$$

Algorithmic typing $\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{x} : \mathtt{T}} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_1 :: *}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 {\to} \mathtt{T}_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_1 \Uparrow (\mathtt{T}_{11} {\to} \mathtt{T}_{12}) \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_2 \leftrightarrow \mathtt{T}_{11}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(TA-APP)}$$

$$\frac{\Gamma, \mathtt{X}{::}\mathtt{K}_1 \mathrel{|\!\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda\mathtt{X}{::}\mathtt{K}_1.\mathtt{t}_2 : \forall\mathtt{X}{::}\mathtt{K}_1.\mathtt{T}_2} \qquad \text{(TA-TABS)}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{T}_1 \Uparrow \forall\mathtt{X}{::}\mathtt{K}_1.\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_1}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{t}_1\ [\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}} \qquad \text{(TA-TAPP)}$$

27.1 Definition: The set of **well-formed contexts** is defined as follows:

$$\vdash \emptyset \text{ ok}$$

$$\frac{\vdash \Gamma \text{ ok} \qquad \Gamma \vdash \mathtt{T} :: *}{\vdash \Gamma, \mathtt{x}{:}\mathtt{T} \text{ ok}}$$

$$\frac{\vdash \Gamma \text{ ok}}{\vdash \Gamma, \mathtt{X}{::}\mathtt{K} \text{ ok}}$$

□

27.2 Proposition: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\vdash \Gamma$ ok, then $\Gamma \vdash \mathtt{T} :: *$. □

Proof: □

27.3 Proposition: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$, then every type annotation on a λ-abstraction within t″ has kind $*$ (in the evident context). □

Proof: Straightforward induction on typing derivations. □

27.4 Theorem [Equivalence of declaractive and algorithmic typing]: Suppose we have $\vdash \Gamma$ ok. Then:

1. If $\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{t} : \texttt{T}$, then $\Gamma \vdash \texttt{t} : \texttt{T}$.
2. If $\Gamma \vdash \texttt{t} : \texttt{T}$, then $\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{t} : \texttt{S}$ with $\Gamma \vdash \texttt{S} \leftrightarrow \texttt{T}$.

□

Proof: □

27.5 Theorem [Decidability]:

1. The context well-formedness rules are syntax-directed and total.
2. The algorithmic typing relation is total as long as the input context is well formed.

□

Proof: □

Chapter 28

Higher-Order Subtyping

Just a sketch.

Some kinding premises are omitted in the subtyping rules presented here...

$F^{\omega}_{<:}$: Higher-order bounded quantification $F_{<:}$ + F^{ω}

Syntax

`t ::=`		*terms*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`λX<:T.t`	*type abstraction*
	`t [T]`	*type application*
`v ::=`		*values*
	`λx:T.t`	*abstraction value*
	`λX<:T.t`	*type abstraction value*
`T ::=`		*types*
	`X`	*type variable*
	`Top`	*maximum type*
	`T→T`	*type of functions*
	`∀X<:T.T`	*universal type*
	`λX::K.T`	*operator abstraction*
	`T T`	*operator application*
`Γ ::=`		*contexts*
	`∅`	*empty context*
	`Γ, x:T`	*term variable binding*

	$\Gamma, \mathtt{X} \mathtt{<:T}$	*type variable binding*
$\mathtt{K}$ ::=		*kinds*
	$*$	*kind of proper types*
	$\mathtt{K} \Rightarrow \mathtt{K}$	*kind of operators*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_{12} \qquad \text{(E-BETA)}$$

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

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-APP2)}$$

$$(\lambda \mathtt{X}\mathtt{<:}\mathtt{T}_{11}.\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{t}_{12} \qquad \text{(E-BETA2)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1'\ [\mathtt{T}_2]} \qquad \text{(E-TAPP)}$$

Type equivalence $\boxed{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{S} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{U} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \leftrightarrow \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-TRANS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 \leftrightarrow \mathtt{T}_1 \to \mathtt{T}_2 :: *} \qquad \text{(Q-ARROW)}$$

$$\frac{\Gamma, \mathtt{X}\mathtt{<:}\mathtt{U} \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \forall \mathtt{X}\mathtt{<:}\mathtt{U}.\mathtt{S}_2 \leftrightarrow \forall \mathtt{X}\mathtt{<:}\mathtt{U}.\mathtt{T}_2 :: *} \qquad \text{(Q-ALL)}$$

$$\frac{\Gamma, \mathtt{X}{::}\mathtt{K}_{11} \vdash \mathtt{T}_{12} :: \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash (\lambda \mathtt{X}{::}\mathtt{K}_{11}.\mathtt{T}_{12})\ \mathtt{T}_2 \leftrightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_{12} :: \mathtt{K}_{12}} \qquad \text{(Q-BETA)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{S} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{T} :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2 \\ \Gamma, \mathtt{X}{::}\mathtt{K}_1 \vdash \mathtt{S}\ \mathtt{X} \leftrightarrow \mathtt{T}\ \mathtt{X} :: \mathtt{K}_2\end{array}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} : \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \qquad \text{(Q-EXT)}$$

$$\frac{\Gamma, \mathtt{X}{::}\mathtt{K}_1 \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda \mathtt{X}{::}\mathtt{K}_1.\mathtt{S}_2 \leftrightarrow \lambda \mathtt{X}{::}\mathtt{K}_1.\mathtt{T}_2 :: \mathtt{K}_1 \Rightarrow \mathtt{K}_2} \qquad \text{(Q-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: \mathtt{K}_{11} \Rightarrow \mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{S}_1\ \mathtt{S}_2 \leftrightarrow \mathtt{T}_1\ \mathtt{S}_2 :: \mathtt{K}_{12}} \qquad \text{(Q-APP)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X}{<:}\mathtt{T} \in \Gamma \qquad \Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \qquad \text{(K-TVar)}$$

$$\frac{\Gamma, \mathtt{X}{<:}\mathtt{T_1} \vdash \mathtt{T_2} :: *}{\Gamma \vdash \forall \mathtt{X}{<:}\mathtt{T_1}.\mathtt{T_2} :: *} \qquad \text{(K-All)}$$

$$\frac{\Gamma, \mathtt{X}{<:}\mathtt{Top}_{\mathtt{K_1}} \vdash \mathtt{T_2} :: \mathtt{K_2}}{\Gamma \vdash \lambda \mathtt{X}{::}\mathtt{K_1}.\mathtt{T_2} :: \mathtt{K_1}{\Rightarrow}\mathtt{K_2}} \qquad \text{(K-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{T_1} :: \mathtt{K_{11}}{\Rightarrow}\mathtt{K_{12}} \qquad \Gamma \vdash \mathtt{T_2} :: \mathtt{K_{11}}}{\Gamma \vdash \mathtt{T_1}\ \mathtt{T_2} :: \mathtt{K_{12}}} \qquad \text{(K-TApp)}$$

$$\frac{\Gamma \vdash \mathtt{T_1} :: * \qquad \Gamma \vdash \mathtt{T_2} :: *}{\Gamma \vdash \mathtt{T_1}{\rightarrow}\mathtt{T_2} :: *} \qquad \text{(K-Arrow)}$$

Subtyping $\boxed{\Gamma \vdash \mathtt{S} <: \mathtt{T}}$

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

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \qquad \text{(S-Eqv)}$$

$$\Gamma \vdash \mathtt{S} <: \mathtt{Top} \qquad \text{(S-Top)}$$

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \mathtt{S_1}{\rightarrow}\mathtt{S_2} <: \mathtt{T_1}{\rightarrow}\mathtt{T_2}} \qquad \text{(S-Arrow)}$$

$$\frac{\mathtt{X}{<:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{X} <: \mathtt{T}} \qquad \text{(S-TVar)}$$

$$\frac{\Gamma, \mathtt{X}{<:}\mathtt{U_1} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \forall \mathtt{X}{<:}\mathtt{U_1}.\mathtt{S_2} <: \forall \mathtt{X}{<:}\mathtt{U_1}.\mathtt{T_2}} \qquad \text{(S-All)}$$

$$\frac{\Gamma, \mathtt{X}{::}\mathtt{K} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{X}{::}\mathtt{K}.\mathtt{S_2} <: \lambda \mathtt{X}{::}\mathtt{K}.\mathtt{T_2}} \qquad \text{(S-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{S_1} <: \mathtt{T_1}}{\Gamma \vdash \mathtt{S_1}\ \mathtt{U} <: \mathtt{T_1}\ \mathtt{U}} \qquad \text{(S-App)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

$$\frac{\Gamma \vdash \mathtt{T_1} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T_1}.\mathtt{t_2} : \mathtt{T_1} \rightarrow \mathtt{T_2}} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}} \rightarrow \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \tag{T-App}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \tag{T-Sub}$$

$$\frac{\Gamma, \mathtt{X} <: \mathtt{T} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{X} <: \mathtt{T}.\mathtt{t_2} : \forall \mathtt{X} <: \mathtt{T}.\mathtt{T_2}} \tag{T-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \forall \mathtt{X} <: \mathtt{T_{11}}.\mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{T_2} <: \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ [\mathtt{T_2}] : \{\mathtt{X} \mapsto \mathtt{T_2}\}\mathtt{T_{12}}} \tag{T-TApp}$$

Abbreviations

$$\text{unannotated binder for } \mathtt{X} \stackrel{\text{def}}{=} \mathtt{X} <: \mathtt{Top}$$

$$\mathtt{Top}_* \stackrel{\text{def}}{=} \mathtt{Top}$$

$$\mathtt{Top}_{\mathtt{K_1} \Rightarrow \mathtt{K_2}} \stackrel{\text{def}}{=} \lambda \mathtt{X} :: \mathtt{K_1}.\ \mathtt{Top}_{\mathtt{K_2}}$$

Chapter 29

Polymorphic Update

This chapter is just a sketch.

The mechanism of polymorphic update is quite a bit less "mainstream" than almost all of what's covered in the rest of the chapters. It's not completely clear that it belongs. On the other hand, it makes the object encoding examples in Chapter 30 vastly clearer.

In fact, I'm tempted to go even further and introduce an ad-hoc "in place update" operation for existential types (following a suggestion in [HP98]). This makes it possible to do the whole of Chapter 30 in a second-order setting, before discussing type operators.

Polymorphic update $\qquad F_{<:} + \{\}$ *update*

New syntactic forms

$\mathtt{t}$	::=	...	*terms*
		$\{\iota_i\, \mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t}\leftarrow\mathtt{l=t}$	*field update*
$\mathtt{T}$	::=	...	*types*
		$\{\iota_i\, \mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*
ι	::=	#	*invariant (updatable) field*
		omitted	*covariant (fixed) field*

New evaluation rules $\qquad \boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\begin{array}{c}\{\iota_i \mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\leftarrow\mathtt{l}_i\mathtt{=v} \\ \longrightarrow \{\iota_i \mathtt{l}_i\mathtt{=v}_i\ ^{j\in 1..i-1}, \mathtt{l}_i\mathtt{=v}, \iota_k \mathtt{l}_k\mathtt{=v}_k\ ^{k\in i+1..n}\}\end{array} \qquad \text{(E-UpdateBeta)}$$

$$\{\iota_i\, \mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-RcdBeta)}$$

$$\frac{\mathtt{t_j} \longrightarrow \mathtt{t_j'}}{\{\iota_i\,\mathtt{l_i}=\mathtt{v_i}\;^{i\in 1..j-1},\iota_j\,\mathtt{l_j}=\mathtt{t_j},\iota_k\,\mathtt{l_k}=\mathtt{t_k}\;^{k\in j+1..n}\} \longrightarrow \{\iota_i\,\mathtt{l_i}=\mathtt{v_i}\;^{i\in 1..j-1},\iota_j\,\mathtt{l_j}=\mathtt{t_j'},\iota_k\,\mathtt{l_k}=\mathtt{t_k}\;^{k\in j+1..n}\}}$$

(E-RECORD)

New subtyping rules $\boxed{\Gamma \vdash \mathtt{S} <: \mathtt{T}}$

$$\Gamma \vdash \{\iota_i\,\mathtt{l_i}:\mathtt{T_i}\;^{i\in 1..n+k}\} <: \{\iota_i\,\mathtt{l_i}:\mathtt{T_i}\;^{i\in 1..n}\}$$ (S-RCD-WIDTH)

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{S_i} <: \mathtt{T_i} \qquad \text{if } \iota_i = \#\text{, then } \Gamma \vdash \mathtt{T_i} <: \mathtt{S_i}}{\Gamma \vdash \{\iota_i\,\mathtt{l_i}:\mathtt{S_i}\;^{i\in 1..n}\} <: \{\iota_i\,\mathtt{l_i}:\mathtt{T_i}\;^{i\in 1..n}\}}$$ (S-RCD-DEPTH)

$$\Gamma \vdash \{\ldots\#\mathtt{l_i}:\mathtt{S_i}\ldots\} <: \{\ldots\mathtt{l_i}:\mathtt{S_i}\ldots\}$$ (S-RCD-VARIANCE)

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t_i} : \mathtt{T_i}}{\Gamma \vdash \{\iota_i\,\mathtt{l_i}=\mathtt{t_i}\;^{i\in 1..n}\} : \{\iota_i\,\mathtt{l_i}:\mathtt{T_i}\;^{i\in 1..n}\}}$$ (T-RCD)

$$\frac{\Gamma \vdash \mathtt{t_0} : \{\iota_i\,\mathtt{l_i}:\mathtt{T_i}\;^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t_0.l_j} : \mathtt{T_j}}$$ (T-PROJ)

$$\frac{\Gamma \vdash \mathtt{r} : \mathtt{R} \qquad \Gamma \vdash \mathtt{R} <: \{\#\mathtt{l_j}:\mathtt{T_j}\} \qquad \Gamma \vdash \mathtt{t} : \mathtt{T_j}}{\Gamma \vdash \mathtt{r}\leftarrow\mathtt{l_j}=\mathtt{t} : \mathtt{R}}$$ (T-UPDATE)

Chapter 30

Purely Functional Objects and Classes

Technically complete, more or less. Work needed on writing.

30.1 Simple Objects

Recall (from Chapter 19) the type of pure (or "purely functional") `Counter` objects:

```
Counter = {∃X, {state:X, methods:{get: X→Nat, inc:X→X}}};
```

The representation type of counters:

```
CounterR = {x:Nat};
```

A counter object:

```
c = {∃X = Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
    as Counter;
```

```
▶ c : Counter
```

Message-sending operations:

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
```

```
▶ sendget : Counter → Nat
```

```
sendinc = λc:Counter.
            let {X,body} = c in
              {∃X = X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter

addthree = λc:Counter. sendinc (sendinc (sendinc c));
▶ addthree : Counter → Counter

c3 = addthree c;
sendget c3;
▶ c3 : Counter
8 : Nat
```

30.2 Subtyping

Recall (from Section **??**) the subtyping rule for existential types:

This means that if we define an object type with more methods than our `Counter` type

```
ResetCounter =
  {∃X, {state:X, methods:{get: X→Nat, inc:X→X, reset:X→X}}};
```

we will have:

`ResetCounter` <: `Counter`

So if we define a reset counter object

```
rc = {∃X = Nat,
      {state = 0,
       methods = {get = λx:Nat. x,
                  inc = λx:Nat. succ(x),
                  reset = λx:Nat. 0}}}
     as ResetCounter;
▶ rc : ResetCounter
```

we can pass it as a legal argument to `sendget` and `sendinc`, and hence also `addthree`:

```
rc3 = addthree rc;
sendget rc3;
▶ rc3 : Counter
3 : Nat
```

Notice, though, that the type of `rc3` here is just `Counter` – passing it through the `addthree` operation has lost some information about its type.

30.3 Interface Types

Using type operators, we rewrite the type `Counter` in two parts,

```
Counter = Object CounterM;
```

where

```
CounterM = λR. {get: R→Nat, inc:R→R};
▸ CounterM = λR. {get:R→Nat, inc:R→R}: * ⇒ *
```

is a type operator representing the interface of counter objects (the part that is specific to the fact that these are counters and not some other kind of objects) and

```
Object = λI::*⇒*. {∃X, {state:X, methods:(I X)}};
▸ Object = λI::*⇒*. {∃X, {state:X,methods:I X}}: (*⇒*) ⇒ *
```

is a type operator (whose parameter is a type operator!) that captures the common structure of all object types.

If we similarly define

```
ResetCounterM = λR. {get: R→Nat, inc:R→R, reset:R→R};
ResetCounter = Object ResetCounterM;
▸ ResetCounterM = λR. {get:R→Nat, inc:R→R, reset:R→R}: * ⇒ *
  ResetCounter = Object ResetCounterM: *
```

we have not only

`ResetCounter` <: `Counter`

as before but also

`ResetCounterM` <: `CounterM`

by the rules above for subtyping between type operators.

30.4 Sending Messages to Objects

We can now give `sendinc` a more refined typing by abstracting over sub-operators of `CounterM`

```
sendinc =
  λM<:CounterM.
  λo:Object(M).
    let {X, b} = o in
      ({∃X = X,
       {state = b.methods.inc(b.state),
        methods = b.methods}}
       as Object(M));
```

```
▶ sendinc : ∀M<:CounterM. Object M → Object M
```

and send messages to counter and reset-counter objects like this (assuming `sendget` and `sendreset` are defined analogously):

```
   sendget [CounterM] (sendinc [CounterM] c);
▶ 6 : Nat

   sendget [ResetCounterM]
              (sendreset [ResetCounterM]
                 (sendinc [ResetCounterM] rc));
▶ 0 : Nat
```

Intuitively, the type of `sendinc` can be read "give me an object interface refining the interface of counters, then give me an object with that interface, and I'll return you another object with the same interface."

30.4.1 Exercise: Check carefully that `sendinc` has the claimed type and that the expressions involving applications of `sendinc` are well typed. □

30.4.2 Exercise: Define `sendget` and `sendreset`. □

30.5 Simple Classes

In Chapter 14, we defined a class to be a function from states to objects, where objects were records of methods. Here, an object is more than just a record of methods: it includes a representation type and a state as well as methods. Moreover, each of the methods here takes the state as a parameter, so it doesn't make sense to pass the state to the class. In fact, a class here (as long as we are holding the representation type fixed) is just a record of methods – no packaging, and no abstraction of the whole thing on the state.

```
   CounterR = Nat;
   counterClass =
      {get = λx:CounterR. x,
       inc = λx:CounterR. succ(x)}
    as {get: CounterR→Nat, inc:CounterR→CounterR};
▶ counterClass : {get:CounterR→Nat, inc:CounterR→CounterR}
```

Or, using the `CounterM` operator to write the annotation more tersely:

```
   counterClass =
      {get = λx:CounterR. x,
       inc = λx:CounterR. succ(x)}
    as CounterM CounterR;
```

```
▶ counterClass : CounterM CounterR
```

We then build an instance by adding state and packaging:

```
c = {∃X = CounterR,
     {state = 0,
      methods = counterClass}}
 as Counter;
```

```
▶ c : Counter
```

Here is a simple subclass...

```
resetCounterClass =
   let super = counterClass in
   {get = super.get,
    inc = super.inc,
    reset = λx:CounterR. 0}
 as ResetCounterM CounterR;
```

```
▶ resetCounterClass : ResetCounterM CounterR
```

30.6 Adding Instance Variables

A record representation for counters:

```
CounterR = {x:Nat};
counterClass =
   {get = λs:CounterR. s.x,
    inc = λs:CounterR. {x=succ(s.x)}}
 as CounterM CounterR;
```

A more interesting representation using updatable records (Chapter 29):

```
CounterR = {#x:Nat};
counterClass =
   {get = λs:CounterR. s.x,
    inc = λs:CounterR. s←x=succ(s.x)}
 as CounterM CounterR;
```

```
▶ counterClass : CounterM CounterR
  counterClass : CounterM CounterR
```

(Note that objects built from these classes will have the same old type `Counter`, and the same message-sending code as before will apply without change.)

Now, because we've used polymorphic update operations to define the methods of the counter class, it is actually not necessary to know that the state has exactly type `CounterR`: we can also work with states that are subtypes of this type.

```
counterClass =
   λR<:CounterR.
      {get = λs:R. s.x,
       inc = λs:R. s←x=succ(s.x)}
   as CounterM R;
▶ counterClass : ∀R<:CounterR. CounterM R
```

To build an object from the new `counterClass`, we simply supply `CounterR` itself as the representation:

```
c = {∃X = CounterR,
     {state = {#x=0},
      methods = counterClass [CounterR]}}
    as Counter;
▶ c : Counter
```

We can write `resetCounterClass` in the same style:

```
resetCounterClass =
   λR<:CounterR.
     let super = counterClass [R] in
       {get = super.get,
        inc = super.inc,
        reset = λs:R. s←x=0}
       as ResetCounterM R;
▶ resetCounterClass : ∀R<:CounterR. ResetCounterM R
```

Finally, we can write `backupCounterClass` in the same style, this time abstracting over a subtype of `BackupCounterR`: We can define another subclass – backup counters – this time using a *different* representation type:

```
BackupCounterM = λR. {get: R→Nat, inc:R→R, reset:R→R, backup:R→R};
BackupCounterR = {#x:Nat,#old:Nat};
backupCounterClass =
   λR<:BackupCounterR.
   let super = resetCounterClass [R] in
    {get = super.get,
     inc = super.inc,
     reset = λs:R. s←x=s.old,
     backup = λs:R. s←old=s.x}
    as BackupCounterM R;
▶ backupCounterClass : ∀R<:BackupCounterR. BackupCounterM R
```

30.6.1 Exercise [Moderate]: We have used polymorphic update in this section to allow methods in classes to update "their" instance variables while leaving room for subclasses to extend the state with additional instance variables. Show that it is possible to achieve the same effect even in a type system lacking polymorphic record update. □

30.7 Classes with "Self"

In Section 14.6, we saw how to extend imperative classes with a mechanism allowing the methods of a class to refer to each other recursively. This extension also makes sense in the pure setting.

We begin by abstracting `counterClass` on a collection of methods (appropriate for the same representation type `R`):

```
counterClass =
   λR<:CounterR.
   λself: CounterM R.
      {get = λs:R. s.x,
       inc = λs:R. s←x=succ(s.x)}
   as CounterM R;
```

To build an object from this class, we have to build the fixed point of the function `counterClass`:

```
c = {∃X = CounterR,
     {state = {#x=0},
      methods = fix (counterClass [CounterR])}}
    as Counter;
```

```
▶ c : Counter
```

To use this new mechanism, we define a new form of "counters with a `set` operation":

```
SetCounterM = λR. {get: R→Nat, set:R→Nat→R, inc:R→R};
```

The implementation of the `setCounterClass` defines a `set` method and uses the `set` and `get` methods of `self` to implement the `inc` method:

```
setCounterClass =
   λR<:CounterR.
   λself: SetCounterM R.
      let super = counterClass [R] self in
      {get = super.get,
       set = λs:R. λn:Nat. s←x=n,
       inc = λs:R. self.set s (succ(self.get s))}
   as SetCounterM R;
```

Finally, we can further extend `setCounterClass` to form a class of instrumented counters, whose `set` operation counts the number of times that it has been called:

```
InstrumentedCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat};
InstrumentedCounterR = {#x:Nat,#count:Nat};
instrumentedCounterClass =
```

```
λR<:InstrumentedCounterR.
λself: InstrumentedCounterM R.
   let super = setCounterClass [R] self in
   {get = super.get,
    set = λs:R. λn:Nat.
            let r = super.set s n in
            r←count=succ(r.count),
    inc = super.inc,
    accesses = λs:R. s.count}
as InstrumentedCounterM R;
```

Note that calls to `inc` are included in the access count, since `inc` is implemented in terms of `self.set`.

30.8 Class Types

We can write the type of `counterClass` as

```
CounterClass = Class CounterM CounterR;
```

where:

```
  Class = λM::*⇒*. λR. ∀R<:R. M R → M R;
▶ Class = λM::*⇒*. λR. ∀R'<:R. M R' → M R': (*⇒*) ⇒ * ⇒ *
```

In other words, `Class M R` is the type of classes with representation type `R` and methods `M`. Class itself is a higher-order operator, like `Object`.

30.9 Generic Inheritance

Going further along the same lines, we can actually use type operators to construct well-typed *terms* that perform instantiation of classes (object creation) and subclassing.

The code that performs the instantiation of `counterClass` to yield a counter does not depend in any way on the fact that we are building a counter as opposed to some other kind of object. We can avoid writing this boilerplate it over and over by abstracting out `counterClass`, `CounterM`, `CounterR`, and the initial value `{#x=0}` as follows:

```
new =
  λM<:TopM.
  λR<:Top.
  λc: Class M R.
  λr:R.
   {∃X = R,
     {state = r,
      methods = fix (c [R])}}
    as Object M;
```

```
▶ new : ∀M<:TopM. ∀R. Class M R → R → Object M
```

where we write `TopM` for the "maximal interface" `λX.Top`:

```
TopM = λX.Top;
```

We can now build counter objects by applying `new` to `counterClass` as follows:

```
c = new [CounterM] [CounterR] counterClass {#x=0};
▶ c : Object CounterM
```

Similarly, we can write a generic function that builds a subclass given a class and a "delta" that shows how the methods of the subclass are derived from the methods of the superclass. We begin with the concrete subclass `setCounterClass` from Section 30.6:

```
setCounterClass =
   λR<:CounterR.
   λself: SetCounterM R.
      let super = counterClass [R] self in
      {get = super.get,
       set = λs:R. λn:Nat. s←x=n,
       inc = λs:R. self.set s (succ(self.get s))}
   as SetCounterM R;
```

Abstracting out the parts that are specific to counters leaves the following parameterized skeleton:

```
extend =
   λSuperM <: TopM.
   λSuperR.
   λsuperClass: Class SuperM SuperR.
   λSelfM <: SuperM.
   λSelfR <: SuperR.
   λdelta: (∀R<:SelfR. SuperM R → SelfM R → SelfM R).
    (λR<:SelfR.
     λself: SelfM R.
       let super = superClass [R] self in
         delta [R] super self)
    as Class SelfM SelfR;
▶ extend : ∀SuperM<:TopM.
             ∀SuperR.
               Class SuperM SuperR →
               (∀SelfM<:SuperM.
                  ∀SelfR<:SuperR.
                    (∀R<:SelfR. SuperM R→SelfM R→SelfM R) →
                    Class SelfM SelfR)
```

In other words, `extend` takes several parameters (`SuperM` to `delta`) and returns a class—i.e., a function that takes a representation type `R` (the "final type" that will eventually be provided by the `new` function) and a vector of "self methods" specialized to the type `R` and returns a new vector of self methods. The most important bit of this definition—the bit that does all the work—is the parameter `delta`. It takes the final type `R` and two method vectors—the "super methods" calculated by instantiating the superclass at `R` and `self` and the "self methods" provided by `new`—and returns the self methods.

Now we can recover `setCounterClass` by extending `counterClass` like this:

```
setCounterClass =
   extend [CounterM] [CounterR] counterClass
          [SetCounterM] [CounterR]
      (λR<:CounterR.
       λsuper: CounterM R.
       λself: SetCounterM R.
         {get = super.get,
          set = λs:R. λn:Nat. s←x=n,
          inc = λs:R. self.set s (succ(self.get s))});
```

```
▶ setCounterClass : Class SetCounterM CounterR
```

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

1. Reimplement `setCounterClass` and `instrumentedCounterClass` using the generic inheritance operations explored in this section.

2. Extend your modified `instrumentedCounterClass` with a subclass that adds a `reset` method. □

Chapter 31

Structures and Modules

By Robert Harper and Benjamin Pierce

It's not clear yet whether this sketch will eventually be just a chapter of the present book or a whole book in its own right. The latter seems more likely at the moment.

31.1 Basic Structures

Structures $\rightarrow$ X $\mathcal{S}$

Syntax

`t ::=`		*terms*
	`t as T`	*coercion*
	`x`	*variable*
	`λx:T.t`	*abstraction*
	`t t`	*application*
	`{`$\mathcal{S}$` X=T::K,t}`	*structure creation*
	`bodyof(t)`	*term component of a structure*
`v ::=`		*values*
	`λx:T.t`	*abstraction value*
	`{`$\mathcal{S}$` X=T::K,v}`	*structure value*
`T ::=`		*types*
	`X`	*type variable*
	`T→T`	*type of functions*
	`{`$\mathcal{S}$` X::K,T}`	*structure type*
	`typeof(t)`	*type component of a structure*
`Γ ::=`		*contexts*

		$\emptyset$	*empty context*
		$\Gamma, x{:}T$	*term variable binding*
		$\Gamma, X{::}K$	*type variable binding*
K	::=		*kinds*
		$*$	*kind of proper types*

Evaluation $\boxed{t \longrightarrow t'}$

$$t_1 \text{ as } T_2 \longrightarrow t_1 \qquad \text{(E-COERCE1)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

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

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

$$\frac{t_1 \longrightarrow t_1'}{\text{bodyof}(t_1) \longrightarrow \text{bodyof}(t_1')} \qquad \text{(E-BODYOF)}$$

$$\frac{X \notin FV(v_2)}{\text{bodyof}(\{\mathcal{S}\,X{=}T_1,v_2\}) \longrightarrow v_2} \qquad \text{(E-STRUCTBETA)}$$

$$\frac{X \in FV(t_2)}{\{\mathcal{S}\,X{=}T_1,t_2\} \longrightarrow \{\mathcal{S}\,X{=}T_1,\{X \mapsto T_1\}t_2\}} \qquad \text{(E-STRUCTSUBST)}$$

$$\frac{X \notin FV(t_2) \qquad t_2 \longrightarrow t_2'}{\{\mathcal{S}\,X{=}T_1,t_2\} \longrightarrow \{\mathcal{S}\,X{=}T_1,t_2'\}} \qquad \text{(E-STRUCT)}$$

Type equivalence $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \leftrightarrow T :: K} \qquad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash T \leftrightarrow S :: K}{\Gamma \vdash S \leftrightarrow T :: K} \qquad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash S \leftrightarrow U :: K \qquad \Gamma \vdash U \leftrightarrow T :: K}{\Gamma \vdash S \leftrightarrow T :: K} \qquad \text{(Q-TRANS)}$$

$$\frac{\Gamma \vdash S_1 \leftrightarrow T_1 :: * \qquad \Gamma \vdash S_2 \leftrightarrow T_2 :: *}{\Gamma \vdash S_1 \to S_2 \leftrightarrow T_1 \to T_2 :: *} \qquad \text{(Q-ARROW)}$$

$$\frac{\Gamma \vdash T_1 :: K_1 \qquad \Gamma \vdash \{X \mapsto T_1\}t_2 : T_2}{\Gamma \vdash \text{typeof}(\{\mathcal{S}\,X{=}T_1,t_2\}) \leftrightarrow T_1 :: K_1} \qquad \text{(Q-STRUCTBETA)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{T}_1 \to \mathtt{T}_2 :: *} \qquad \text{(K-ARROW)}$$

$$\frac{\mathtt{X} :: \mathtt{K} \in \Gamma}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \qquad \text{(K-TVAR)}$$

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K} \vdash \mathtt{T}_1 :: *}{\Gamma \vdash \{\mathcal{S}\,\mathtt{X} :: \mathtt{K}, \mathtt{T}_1\} :: *} \qquad \text{(K-STRUCT)}$$

$$\frac{\Gamma \vdash \mathtt{v}_1 : \{\mathcal{S}\,\mathtt{X} :: \mathtt{K}, \mathtt{T}_1\}}{\Gamma \vdash \mathtt{typeof}(\mathtt{v}_1) :: \mathtt{K}} \qquad \text{(K-TYPEOF)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

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

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

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

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-EQ)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \mathtt{K}_1 \qquad \Gamma \vdash \{\mathtt{X} \mapsto \mathtt{T}_1\}\mathtt{t}_2 : \{\mathtt{X} \mapsto \mathtt{T}_1\}\mathtt{T}_2 \qquad \Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \{\mathcal{S}\,\mathtt{X} = \mathtt{T}_1, \mathtt{t}_2\} : \{\mathcal{S}\,\mathtt{X} :: \mathtt{K}_1, \mathtt{T}_2\}} \qquad \text{(T-STRUCT)}$$

$$\frac{\Gamma \vdash \mathtt{v}_0 : \{\mathcal{S}\,\mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2\}}{\Gamma \vdash \mathtt{bodyof}(\mathtt{v}_0) : \{\mathtt{X} \mapsto \mathtt{typeof}(\mathtt{v}_0)\}\mathtt{T}_2} \qquad \text{(T-BODYOF)}$$

Abbreviations

unannotated binder for $\mathtt{X} \quad \stackrel{\text{def}}{=} \quad \mathtt{X} :: *$

Important points:

- The calculus relies on an evaluation relation to determine under what circumstances `typeof(t)` makes sense. In the presence of effects, it need not make sense.

- We omit a general "valuability" judgement at this point, in favor of a syntactic definition of "open values"
- `typeof` always makes sense on values (or more generally, later, a valuable expression)
- `bodyof` is restricted to values because of the substitution of `typeof`. (We could go a little further and allow it for nonvalues in the non-dependent case, where the substitution doesn't do anything.)
- With these restrictions, all of this should be encodable using existentials with the `open` elimination form.
- We don't evaluate type expressions, since they play no role in computation. This would need to be revisited if we had any kind of type analysis (`Dynamic`, etc.).
- Structure values are binders. Dependencies are eliminated during projection (this is easy in the binary case).
- On the face of it, there are two choices for type equivalence for `typeof`: (1) we can include the computation rule above for reducing a typeof applied to a structure, or (2) we can eliminate this rule and rely on reflexivity for comparing typeof(structure) expressions. We have chosen the first because it supports a "phase distinction," in the sense that type equality never depends on any notion of equality for ordinary expressions — i.e. type equality can be computed by looking only at the types. Taking the second choice is a step on the road to structure sharing.
- We have omitted the congruence rule for type reduction for `typeof`, since the only possible arguments to `typeof` are already values (when we extend to paths later, this needs to be revisited — note that this may be tricky, since we do not have a "full beta-equivalence" notion of equality on terms: we may need to define a simplification relation separate from evaluation)
- Structure values do not have principal types, for the same reason as existential values did not. The fix is the same: if every structure intro form is immediately ascribed, then the term will have a principal type.

31.2 Record Kinds and Subkinding

Records of types $\rightarrow \Rightarrow$ *trcd*

New syntactic forms

$\mathtt{T}$	::=	...	*types*
		$\{\mathtt{l}_i\mathtt{=T}_i\ ^{i\in 1..n}\}$	*record of types*
		$\mathtt{T.l}$	*projection*
$\mathtt{K}$	::=	...	*kinds*
		$\{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\}$	*kind of records of types*

New type equivalence rules $\boxed{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{K}_1 <: \mathtt{K}_2}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}_2} \tag{Q-Sub}$$

$$\frac{\Gamma \vdash\ ^{i\in 1..n}\mathtt{T}_i :: \mathtt{K}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=T}_i\ ^{i\in 1..n}\}.\mathtt{l}_i \leftrightarrow \mathtt{T}_i :: \mathtt{K}_i} \tag{Q-TRcdBeta}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \leftrightarrow \mathtt{T}_1' :: \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{T}_1.\mathtt{l}_i \leftrightarrow \mathtt{T}_1'.\mathtt{l}_i :: \mathtt{K}_i} \tag{Q-TyProj}$$

$$\frac{\Gamma \vdash\ ^{i\in 1..n}\mathtt{T}_i \leftrightarrow \mathtt{T}_i' :: \mathtt{K}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=T}_i\ ^{i\in 1..n}\} \leftrightarrow \{\mathtt{l}_i\mathtt{=T}_i'\ ^{i\in 1..n}\} :: \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\}} \tag{Q-TRcd}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\} \qquad \Gamma \vdash \mathtt{T}' :: \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\} \qquad \Gamma \vdash\ ^{i\in 1..n}\mathtt{T}.\mathtt{l}_i \leftrightarrow \mathtt{T}'.\mathtt{l}_i :: \mathtt{K}_i}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T}' :: \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\}} \tag{Q-TRcdExt}$$

$$\frac{\Gamma \vdash \{\mathtt{l}_i\mathtt{=T}_i\ ^{i\in 1..n}\} :: \mathtt{K} \qquad \Gamma \vdash \{\mathtt{l}_{\pi i}\mathtt{=T}_{\pi i}\ ^{i\in 1..n}\} :: \mathtt{K} \qquad \pi \text{ is a permutation of } \{1..n\}}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=T}_i\ ^{i\in 1..n}\} \leftrightarrow \{\mathtt{l}_{\pi i}\mathtt{=T}_{\pi i}\ ^{i\in 1..n}\} :: \mathtt{K}} \tag{Q-TRcd-Perm}$$

New kind formation rules

$$\frac{\Gamma \vdash\ ^{i\in 1..n}\mathtt{K}_i \text{ ok}}{\Gamma \vdash \{\mathtt{l}_i :: \mathtt{K}_i\ ^{i\in 1..n}\} \text{ ok}} \tag{KF-TRcd}$$

New subkinding rules

$$\frac{\Gamma \vdash \mathtt{K}_1 \leftrightarrow \mathtt{K}_2}{\Gamma \vdash \mathtt{K}_1 <: \mathtt{K}_2} \tag{SK-Eqv}$$

$$\frac{\Gamma \vdash K_1 <: K_2 \qquad \Gamma \vdash K_2 <: K_3}{\Gamma \vdash K_1 <: K_3} \tag{SK-TRANS}$$

$$\frac{\Gamma \vdash \{l_i :: K_i \; ^{i \in 1..n+k}\} \text{ ok}}{\Gamma \vdash \{l_i :: K_i \; ^{i \in 1..n+k}\} <: \{l_i :: K_i \; ^{i \in 1..n}\}} \tag{S-TRCD-WIDTH}$$

$$\frac{\Gamma \vdash \{l_i :: J_i \; ^{i \in 1..n}\} \text{ ok} \qquad \Gamma \vdash \{l_i :: K_i \; ^{i \in 1..n}\} \text{ ok} \qquad \Gamma \vdash \; ^{i \in 1..n} J_i <: K_i}{\Gamma \vdash \{l_i :: J_i \; ^{i \in 1..n}\} <: \{l_i :: K_i \; ^{i \in 1..n}\}} \tag{SK-TRCD-DEPTH}$$

New kind equivalence rules

$$\frac{\pi \text{ is a permutation of } \{1..n\} \qquad \Gamma \vdash \{l_i :: K_i \; ^{i \in 1..n}\} \text{ ok} \qquad \Gamma \vdash \{l_{\pi i} :: K_{\pi i}^{i \in 1..n}\} \text{ ok}}{\Gamma \vdash \{l_i :: K_i \; ^{i \in 1..n}\} \leftrightarrow \{l_{\pi i} :: K_{\pi i}^{i \in 1..n}\}} \tag{KEQV-TRCD-PERM}$$

New kinding rules $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma \vdash \; ^{i \in 1..n} T_i :: K_i}{\Gamma \vdash \{l_i = T_i \; ^{i \in 1..n}\} :: \{l_i :: K_i \; ^{i \in 1..n}\}} \tag{K-TRCD}$$

$$\frac{\Gamma \vdash T_0 :: \{l_i :: K_i \; ^{i \in 1..n}\}}{\Gamma \vdash T_0.l_j :: K_j} \tag{K-TYPROJ}$$

New abbreviations

$$\{\ldots K_i \ldots\} \stackrel{\text{def}}{=} \{\ldots i :: K_i \ldots\}$$

Notes:

- We include an extensionality rule for several reasons:
  - It's natural. (Programmers would expect it.)
  - It's needed for selfification to work.
  - The present algorithm relies on it (weak argument, but there it is)

31.3 Singleton Kinds

Singleton kinds $\rightarrow$ Eq

New syntactic forms

`K`	::=	...	*kinds*
		`Eq(T)`	*singleton kind*

New type equivalence rules $\boxed{\Gamma \vdash S \leftrightarrow T}$

$$\frac{\Gamma \vdash S \leftrightarrow T :: K_1 \qquad \Gamma \vdash K_1 <: K_2}{\Gamma \vdash S \leftrightarrow T :: K_2} \quad \text{(Q-SUB)}$$

$$\frac{\Gamma \vdash S :: Eq(T)}{\Gamma \vdash S \leftrightarrow T :: *} \quad \text{(Q-SINGLETON1)}$$

$$\frac{\Gamma \vdash S \leftrightarrow T :: *}{\Gamma \vdash S \leftrightarrow T :: Eq(S)} \quad \text{(Q-SINGLETON2)}$$

New kind formation rules

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash Eq(S) \text{ ok}} \quad \text{(KF-SINGLETON)}$$

New kind equivalence rules

$$\frac{\Gamma \vdash S \leftrightarrow T :: *}{\Gamma \vdash Eq(S) \leftrightarrow Eq(T)} \quad \text{(KEQV-SINGLETON)}$$

New subkinding rules

$$\frac{\Gamma \vdash K_1 \leftrightarrow K_2}{\Gamma \vdash K_1 <: K_2} \quad \text{(SK-EQV)}$$

$$\frac{\Gamma \vdash K_1 <: K_2 \qquad \Gamma \vdash K_2 <: K_3}{\Gamma \vdash K_1 <: K_3} \quad \text{(SK-TRANS)}$$

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash Eq(S) <: *} \quad \text{(SK-SINGLETON)}$$

New kinding rules $\boxed{\Gamma \vdash T :: K}$

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash S :: Eq(S)} \quad \text{(K-SINGLETON)}$$

31.4 Dependent Function and Record Kinds

Dependent records of types $\to$ $\Rightarrow$ *trcd* *depkind*

New syntactic forms

$\mathtt{K}$ ::=	...	*kinds*
	$\Pi\mathtt{X}::\mathtt{K}.\mathtt{K}$	*dependent kind of operators*
	$\{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\}$	*dependent kind of records of types*

New kind formation rules

$$\frac{\Gamma \vdash \mathtt{K}_1 \text{ ok} \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2 \text{ ok}}{\Gamma \vdash \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \text{ ok}} \qquad \text{(KF-TARR)}$$

$$\frac{\Gamma, \mathtt{X}_1::\mathtt{K}_1 \cdots \mathtt{X}_{i-1}::\mathtt{K}_{i-1} \vdash^{i\in 1..n} \mathtt{K}_i \text{ ok}}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\} \text{ ok}} \qquad \text{(KF-TRCD)}$$

New kind equivalence rules

$$\frac{\Gamma \vdash \mathtt{K}_1 \leftrightarrow \mathtt{K}_1' \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{K}_2 \leftrightarrow \mathtt{K}_2'}{\Gamma \vdash \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \leftrightarrow \Pi\mathtt{X}::\mathtt{K}_1'.\mathtt{K}_2'} \qquad \text{(KEQV-TARR)}$$

$$\frac{\begin{array}{c}\pi \text{ is a permutation of } \{1..n\} \\ \Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\} \text{ ok} \\ \Gamma \vdash \{\mathtt{l}_{\pi i} \triangleright \mathtt{X}_{\pi i} :: \mathtt{K}_{\pi i}^{i\in 1..n}\} \text{ ok}\end{array}}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\} \leftrightarrow \{\mathtt{l}_{\pi i} \triangleright \mathtt{X}_{\pi i} :: \mathtt{K}_{\pi i}^{i\in 1..n}\}} \qquad \text{(KEQV-TRCD-PERM)}$$

New kinding rules $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T}_2 :: \mathtt{K}_2}{\Gamma \vdash \lambda\mathtt{X}::\mathtt{K}_1.\mathtt{T}_2 :: \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2} \qquad \text{(K-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: \Pi\mathtt{X}::\mathtt{K}_{11}.\mathtt{K}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{T}_1\ \mathtt{T}_2 :: \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{K}_{12}} \qquad \text{(K-TAPP)}$$

$$\frac{\Gamma, \mathtt{X}_1::\mathtt{K}_1 \cdots \mathtt{X}_{i-1}::\mathtt{K}_{i-1} \vdash^{i\in 1..n} \mathtt{T}_i :: \mathtt{K}_i}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i = \mathtt{T}_i{}^{i\in 1..n}\} :: \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\}} \qquad \text{(K-TRCD)}$$

$$\frac{\Gamma \vdash \mathtt{T}_0 :: \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\}}{\Gamma \vdash \mathtt{T}_0.\mathtt{l}_j :: \{\mathtt{X}_1 \mapsto \mathtt{T}_0.\mathtt{l}_1\}\cdots\{\mathtt{X}_{j-1} \mapsto \mathtt{T}_0.\mathtt{l}_{j-1}\}\mathtt{K}_j} \qquad \text{(K-TYPROJ)}$$

New subkinding rules

$$\frac{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n+k}\} \text{ ok}}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n+k}\} <: \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{i\in 1..n}\}} \qquad \text{(S-TRCD-WIDTH)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{J}_i{}^{\,i\in 1..n}\}\ \text{ok} \\ \Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{\,i\in 1..n}\}\ \text{ok} \\ \Gamma, \mathtt{X}_1 :: \mathtt{J}_1 \cdots \mathtt{X}_{i-1} :: \mathtt{J}_{i-1} \vdash {}^{i\in 1..n}\ \mathtt{J}_i <: \mathtt{K}_i\end{array}}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{J}_i{}^{\,i\in 1..n}\} <: \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i{}^{\,i\in 1..n}\}} \quad \text{(SK-TRCD-DEPTH)}$$

Notes:

- We introduced primitive labeled record kinds because Cardelli's encoding doesn't work in the presence of dependencies, because it presumes the ability to reorder labels arbitrarily to match the fixed global order; but this may break dependencies between fields.
- In K-TRCD, note the nondeterminism in the choice of K's.

31.5 Dependent Record Expressions and Records of Types

Dependent record terms (??) **???**

New syntactic forms

$\mathtt{t}$ ::=	...	*terms*
	$\{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{t}_i{}^{\,i\in 1..n}\}$	*dependent record*
$\mathtt{v}$ ::=	...	*values*
	$\{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{v}_i{}^{\,i\in 1..n}\}$	*dependent record value*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\text{for each } i \quad \{\mathtt{x}_1 \ldots \mathtt{x}_{i-1}\} \cap FV(\mathtt{v}_i) = \emptyset}{\{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{v}_i{}^{\,i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j} \quad \text{(E-RCDBETA)}$$

$$\frac{\begin{array}{c}\text{for each } j < i, \{\mathtt{x}_1 \ldots \mathtt{x}_{j-1}\} \cap FV(\mathtt{v}_j) = \emptyset \\ \text{for some } j < i, \mathtt{x}_j \in FV(\mathtt{t}_i) \\ \mathtt{t'}_i = \{\mathtt{x}_1 \mapsto \mathtt{v}_1\} \cdots \{\mathtt{x}_{i-1} \mapsto \mathtt{v}_{i-1}\}\mathtt{t}_i\end{array}}{\begin{array}{c}\{\mathtt{l}_j \triangleright \mathtt{x}_j = \mathtt{v}_j{}^{\,j\in 1..i-1}, \mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{t}_i, \mathtt{l}_k \triangleright \mathtt{x}_k = \mathtt{t}_k{}^{\,k\in i+1..n}\} \longrightarrow \\ \{\mathtt{l}_j \triangleright \mathtt{x}_j = \mathtt{v}_j{}^{\,j\in 1..i-1}, \mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{t'}_i, \mathtt{l}_k \triangleright \mathtt{x}_k = \mathtt{t}_k{}^{\,k\in i+1..n}\}\end{array}} \quad \text{(E-RECORDSUBST)}$$

$$\frac{\begin{array}{c}\text{for each } i < j, \{\mathtt{x}_1 \ldots \mathtt{x}_{i-1}\} \cap FV(\mathtt{v}_i) = \emptyset \\ \{\mathtt{x}_1 \ldots \mathtt{x}_{j-1}\} \cap FV(\mathtt{t}_j) = \emptyset \\ \mathtt{t}_j \longrightarrow \mathtt{t}'_j\end{array}}{\begin{array}{c}\{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{v}_i{}^{\,i\in 1..j-1}, \mathtt{l}_j \triangleright \mathtt{x}_j = \mathtt{t}_j, \mathtt{l}_k \triangleright \mathtt{x}_k = \mathtt{t}_k{}^{\,k\in j+1..n}\} \longrightarrow \\ \{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{v}_i{}^{\,i\in 1..j-1}, \mathtt{l}_j \triangleright \mathtt{x}_j = \mathtt{t}'_j, \mathtt{l}_k \triangleright \mathtt{x}_k = \mathtt{t}_k{}^{\,k\in j+1..n}\}\end{array}} \quad \text{(E-RECORD)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma, \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_{i-1} : \mathtt{T}_{i-1} \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{x}_i = \mathtt{t}_i\ ^{i \in 1..n}\} : \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i\ ^{i \in 1..n}\}} \qquad \text{(T-RCD)}$$

New abbreviations

$$\{\ldots \mathtt{t}_i \ldots\} \stackrel{\text{def}}{=} \{\ldots \mathtt{i} \triangleright \mathtt{x}_i = \mathtt{t}_i \ldots\} \text{ where } \mathtt{x}_i \text{ is fresh}$$

31.6 Higher-Kind Singletons

Higher-order singletons **????**

New syntactic forms

$\mathtt{K}$	::=	...	*kinds*
		$\mathtt{EQ(T::K)}$	*higher singleton kind*

New kind formation rules

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{EQ(T::K)}\ \text{ok}} \qquad \text{(KF-HOS)}$$

New kind equivalence rules

$$\frac{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T}' :: \mathtt{K} \qquad \Gamma \vdash \mathtt{K} \leftrightarrow \mathtt{K}'}{\Gamma \vdash \mathtt{EQ(T::K)} \leftrightarrow \mathtt{EQ(T'::K')}} \qquad \text{(KEQV-HOS)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathtt{EQ(T::*)} \leftrightarrow \mathtt{Eq(T)}} \qquad \text{(KEQV-HOS-TYPE)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{Eq(U)}}{\Gamma \vdash \mathtt{EQ(T::Eq(U))} \leftrightarrow \mathtt{Eq(T)}} \qquad \text{(KEQV-HOS-SINGLETON)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2}{\Gamma \vdash \mathtt{EQ}(\mathtt{T}::\Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2) \leftrightarrow \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{EQ}(\mathtt{T\ X} :: \mathtt{K}_2)} \qquad \text{(KEQV-HOS-TARR)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i\ ^{i \in 1..n}\}}{\Gamma \vdash \mathtt{EQ}(\mathtt{T}::\{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{K}_i\ ^{i \in 1..n}\}) \leftrightarrow \{\mathtt{l}_i \triangleright \mathtt{X}_i :: \mathtt{EQ}(\mathtt{T.l}_i :: \mathtt{K}_i)\ ^{i \in 1..n}\}}$$

(KEQV-HOS-TRCD)

New kinding rules $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{T} :: \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K}_2 \qquad \Gamma, \mathtt{X}::\mathtt{K}_1 \vdash \mathtt{T\ X} :: \mathtt{K'}_2}{\Gamma \vdash \mathtt{T} :: \Pi\mathtt{X}::\mathtt{K}_1.\mathtt{K'}_2} \qquad \text{(K-TARR-SELF)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \{\mathtt{l_i} \triangleright \mathtt{X_i} :: \mathtt{K_i}\ ^{i \in 1..n}\} \quad \Gamma, \mathtt{X_1} :: \mathtt{K_1} \cdots \mathtt{X_{i-1}} :: \mathtt{K_{i-1}} \vdash^{i \in 1..n} \mathtt{T.l_i} :: \mathtt{K'_i}}{\Gamma \vdash \mathtt{T} :: \{\mathtt{l_i} \triangleright \mathtt{X_i} :: \mathtt{K'_i}\ ^{i \in 1..n}\}} \quad \text{(K-TRCD-SELF)}$$

31.7 First-class Substructures and Functors

Dependent functions **???**

New syntactic forms

$\mathtt{T} ::= \ldots$ *types*

$\Pi\mathtt{x}:\mathtt{T}.\mathtt{T}$ *type of dependent functions*

New type equivalence rules $\boxed{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T} :: \mathtt{K}} \quad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{S} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \quad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{U} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \leftrightarrow \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \quad \text{(Q-TRANS)}$$

$$\frac{\Gamma \vdash \mathtt{S_1} \leftrightarrow \mathtt{T_1} :: * \qquad \Gamma, \mathtt{x}:\mathtt{S_1} \vdash \mathtt{S_2} \leftrightarrow \mathtt{T_2} :: *}{\Gamma \vdash \Pi\mathtt{x}:\mathtt{S_1}.\mathtt{S_2} \leftrightarrow \Pi\mathtt{x}:\mathtt{T_1}.\mathtt{T_2} :: *} \quad \text{(Q-ARROW)}$$

New kinding rules $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{T_1} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{T_2} :: *}{\Gamma \vdash \Pi\mathtt{x}:\mathtt{T_1}.\mathtt{T_2} :: *} \quad \text{(K-ARROW)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T_1} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T_1}.\mathtt{t_2} : \Pi\mathtt{x}:\mathtt{T_1}.\mathtt{T_2}} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \Pi\mathtt{x}:\mathtt{T_{11}}.\mathtt{T_{12}} \qquad \mathtt{x} \notin FV(\mathtt{T_{12}}) \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad \text{(T-EQ)}$$

New abbreviations

$$\mathtt{T_1} \to \mathtt{T_2} \quad \stackrel{\text{def}}{=} \quad \Pi\mathtt{x}:\mathtt{T_1}.\mathtt{T_2} \text{ where } \mathtt{x} \notin FV(\mathtt{T_2})$$

Dependent records **????**

New syntactic forms

$$\begin{array}{llr} \mathtt{T} & ::= \quad \ldots & \textit{types} \\ & \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i{}^{i \in 1..n}\} & \textit{type of dependent records} \end{array}$$

New type equivalence rules $\boxed{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{T} \leftrightarrow \mathtt{S} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-SYMM)}$$

$$\frac{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{U} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \leftrightarrow \mathtt{T} :: \mathtt{K}}{\Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: \mathtt{K}} \qquad \text{(Q-TRANS)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \leftrightarrow \mathtt{T}_1 :: * \qquad \Gamma \vdash \mathtt{S}_2 \leftrightarrow \mathtt{T}_2 :: *}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 \leftrightarrow \mathtt{T}_1 \to \mathtt{T}_2 :: *} \qquad \text{(Q-ARROW)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i{}^{i \in 1..n}\} :: * \\ \Gamma \vdash \{\mathtt{l}_{\pi i} \triangleright \mathtt{x}_{\pi i} : \mathtt{T}_{\pi i}^{i \in 1..n}\} :: * \\ \pi \text{ is a permutation of } \{1..n\} \end{array}}{\Gamma \vdash \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i{}^{i \in 1..n}\} \leftrightarrow \{\mathtt{l}_{\pi i} \triangleright \mathtt{x}_{\pi i} : \mathtt{T}_{\pi i}^{i \in 1..n}\} :: *} \qquad \text{(Q-RCD-PERM)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leftrightarrow \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad \text{(T-EQ)}$$

$$\frac{\text{for each } i \quad \Gamma, \mathtt{x}_1 : \mathtt{T}_1 \cdots \mathtt{x}_{i-1} : \mathtt{T}_{i-1} \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i = \mathtt{t}_i{}^{i \in 1..n}\} : \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i{}^{i \in 1..n}\}} \qquad \text{(T-RCD)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \{\mathtt{l}_i \triangleright \mathtt{x}_i : \mathtt{T}_i{}^{i \in 1..n}\} \qquad \{\mathtt{x}_1 \ldots \mathtt{x}_{j-1}\} \cap FV(\mathtt{T}_j) = \emptyset}{\Gamma \vdash \mathtt{t}_0 . \mathtt{l}_j : \mathtt{T}_j} \qquad \text{(T-PROJ)}$$

New abbreviations

$$\{\ldots \mathtt{T}_i \ldots\} \quad \stackrel{\text{def}}{=} \quad \{\ldots \mathtt{i} \triangleright \mathtt{x} : \mathtt{T}_i \ldots\} \text{ where } \mathtt{x} \text{ is fresh}$$

Notes:

- These types are only apparently dependent: the elim forms apply only in the non-dependent case, in the expectation that dependencies will first be eliminated by propagation of sharing

- This approach depends crucially on the presence of singleton kinds, since these provide the mechanism for propagating sharing
- We hold the type of earlier substructures abstract while checking the next one — i.e., we propagate forward only type information, not "identity"
- The "selfification" rules allow us to specialize signatures (types of modules) to propagate sharing. They are critical.
- Note that the T-RCD rule here does not do any substitution of earlier fields. It could, but this would constitute a violation of abstraction (besides raising the usual issues in combination with effects). This version of the rule, though, relies on the presence of singleton kinds to be very useful. For example, we want the expression

  `{l={`$\mathcal{S}$` Int,...}, l'=3}`

  will not have the intended type

  `{l`$\triangleright$` x:{`$\mathcal{S}$` *,...}, l'`$\triangleright$` x':typeof(x)}`

  by a direct application of this rule. Instead, we need to use singletons to give the term the type

  `{l`$\triangleright$` x:{`$\mathcal{S}$` Eq(Int),...}, l'`$\triangleright$` x':typeof(x)}`

  (a subtype of the previous type), from which T-RCD does derive the intended type.

31.8 Second-Class Modules

phase-separation equations

31.9 Other points to make

- The old argument about whether modules are strong sums.
  - The argument that strong sums are "not abstract" doesn't hold water. Our structures here are not either: the abstraction comes from the combination of structures with ascription (or let-binding, etc.)
  - A better reason is that strong sums propagate sharing information by substitution, which doesn't scale to systems with effects (the Harper/Lillibridge calculus can be described as a variant of strong sums that does this right)
- Applicative vs. generative functors

Chapter 32

Planned Chapters

I'm not sure yet how much material will get added after this point—it depends mostly on how much time it takes to get the rest into a polished state—but here are my top priorities:

- *Basic material on modularity. (Bob Harper and I are working together on a development that includes enough technicalities to understand module systems of the complexity of SML97's (or OCaml's), including both first- and second-class variants. It's a huge task, and it's not clear yet whether we're going to be able to finish something, or whether, if we do, it will turn out to be just a long co-authored chapter here or a short book in its own right.)*
- *Basics of dependent types. (General sums are the point where this gets hard.)*
- *Maybe some elementary material on intersection types, linear types, or the Curry-Howard isomorphism.*
- *A chapter on parametricity and representation independence.*
- *Some material on contextual equivalence. Maybe not a full development, but at least the basic definitions and some examples of where types make a difference. (Some of this is covered in the denotational semantics chapter, to a small extent.)*
- *Some material on translations between calculi, showing how they preserve typing and semantics*
- *A chapter on Moggi's computational lambda-calculus*
- *Something on curry-howard. (In any case, make sure these are in the bibliography: gallier notes on constructive logic)*

Appendices

Appendix A

Solutions to Selected Exercises

Solution to 3.2.15:

$$\frac{\mathtt{t} \longrightarrow \mathtt{t}'}{\mathtt{t} \longrightarrow^* \mathtt{t}'}$$

$$\mathtt{t} \longrightarrow^* \mathtt{t}$$

$$\frac{\mathtt{t} \longrightarrow^* \mathtt{t}' \qquad \mathtt{t}' \longrightarrow^* \mathtt{t}''}{\mathtt{t} \longrightarrow^* \mathtt{t}''}$$

Solution to 3.2.19: By induction on the structure of $\mathtt{t}$.

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

By Proposition 3.2.18, this case cannot occur.

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

Looking at the evaluation rules, we find that only the rule E-SUCC could possibly be used to derive $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathtt{t} \longrightarrow \mathtt{t}''$ (all the other rules have left-hand sides whose outermost constructor is something other than `succ`). So there must be two subderivations with conclusions of the form $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$ and $\mathtt{t}_1 \longrightarrow \mathtt{t}_1''$. By the induction hypothesis (which applies because $\mathtt{t}_1$ is a subterm of $\mathtt{t}$), we obtain $\mathtt{t}_1' = \mathtt{t}_1''$. But then $\mathtt{succ}\ \mathtt{t}_1' = \mathtt{succ}\ \mathtt{t}_1''$, as required.

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

Here there are three evaluation rules (E-PRED, E-BETANATPZ, and E-BETANATPS) that might have been used to reduct $\mathtt{t}$ to $\mathtt{t}'$ and $\mathtt{t}''$. Notice, however, that these rules do not overlap: if $\mathtt{t}$ matches the left-hand side of one rule, then it definitely does *not* match the left-hand side of the others. (For example, if $\mathtt{t}$ matches E-PRED, then $\mathtt{t}_1$ is definitely not a value, in particular not `0` or `succ v`.) This tells us that the *same* rule must have been used to derive $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathtt{t} \longrightarrow \mathtt{t}''$. If that rule was E-PRED, then we use the induction hypothesis as in the previous case. If it was E-BETANATPZ or E-BETANATPS, then the result is immediate.

Case: Other cases
Similar.

Solution to 3.2.24: Let us use the metavariable $\mathtt{t}$ to range over the new set of terms extended with $\mathtt{wrong}$ (including all terms with $\mathtt{wrong}$ as a subphrase), and $\mathtt{g}$ to range over the original set of "good" terms that do not involve wrong. Write $\mathtt{t} \xrightarrow{w} \mathtt{t}'$ for the new evaluation relation augmented with $\mathtt{wrong}$ transitions, and $\mathtt{g} \xrightarrow{o} \mathtt{g}'$ for the original form of evaluation. Now, the claim that the two treatments agree can be formulated precisely as follows: any (original) term whose evaluation gets stuck in the original semantics will evaluate to $\mathtt{wrong}$ in the new semantics, and vice versa. Formally:

A.1 Proposition: For all original terms $\mathtt{g}$,

$$(\mathtt{g} \xrightarrow{o}^{*} \mathtt{g}', \text{ where } \mathtt{g}' \text{ is stuck}) \quad \text{iff} \quad (\mathtt{g} \xrightarrow{w}^{*} \mathtt{wrong}). \qquad \square$$

To prove this, we proceed in several steps. First, we remark that the new transitions we have added do not break Theorem 3.2.19

A.2 Lemma: The augmented evaluation relation is deterministic. $\square$

This means that whenever $\mathtt{g}$ can take a single step to $\mathtt{g}'$ in the original semantics, it can also step to $\mathtt{g}'$ in the augmented semantics, and furthermore that $\mathtt{g}'$ is the *only* term $\mathtt{g}$ can step to in the new semantics.

Next, we show that a term that is (already) stuck in the original semantics will always evaluate to $\mathtt{wrong}$ in the augmented semantics.

A.3 Lemma: If $\mathtt{g}$ is stuck then $\mathtt{g} \xrightarrow{w}^{*} \mathtt{wrong}$. $\square$

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

Case: $\mathtt{g} = \mathtt{true}$, $\mathtt{false}$, or $\mathtt{0}$

Can't happen (we assumed that $\mathtt{g}$ is stuck).

Case: $\mathtt{g} = \mathtt{if}\ \mathtt{g}_1\ \mathtt{then}\ \mathtt{g}_2\ \mathtt{else}\ \mathtt{g}_3$

Since $\mathtt{g}$ is stuck, $\mathtt{g}_1$ must be in normal form (otherwise rule E-IF would apply). Clearly, $\mathtt{g}_1$ cannot be $\mathtt{true}$ or $\mathtt{false}$ (otherwise one of the rules E-BOOLBETAT or E-BOOLBETAF would apply and $\mathtt{g}$ would not be stuck). Consider the remaining cases:

> *Subcase:* $\mathtt{g}_1 = \mathtt{if}\ \mathtt{g}_{11}\ \mathtt{then}\ \mathtt{g}_{12}\ \mathtt{else}\ \mathtt{g}_{13}$
>
> Since $\mathtt{g}_1$ is in normal form and obviously not a value, it is stuck. The induction hypothesis tells us that $\mathtt{g}_1 \xrightarrow{w}^{*} \mathtt{wrong}$. From this, we can construct a derivation of $\mathtt{g} \xrightarrow{w}^{*} \mathtt{if}\ \mathtt{wrong}\ \mathtt{then}\ \mathtt{g}_2\ \mathtt{else}\ \mathtt{g}_3$. Adding a final instance of rule E-IF-WRONG yields $\mathtt{g} \xrightarrow{w}^{*} \mathtt{wrong}$, as required.

Subcase: $\mathtt{g}_1 = \mathtt{succ}\ \mathtt{g}_{11}$

If $\mathtt{g}_{11}$ is a numeric value, then $\mathtt{g}_1$ is a `badbool` and rule E-IF-WRONG immediately yields $\mathtt{g} \xrightarrow{w} \mathtt{wrong}$. Otherwise, by definition $\mathtt{g}_1$ is stuck, and the induction hypothesis tells us that $\mathtt{g}_1 \xrightarrow{w}{}^* \mathtt{wrong}$. From this derivation, we construct a derivation of $\mathtt{g} \xrightarrow{w}{}^* \mathtt{succ\ wrong}$. Adding a final instance of rule E-SUCC-WRONG yields $\mathtt{g} \xrightarrow{w}{}^* \mathtt{wrong}$.

Other subcases:

Similar.

Case: $\mathtt{g} = \mathtt{succ}\ \mathtt{g}_1$

Since g is stuck, we know (from the definition of values) that $\mathtt{g}_1$ must be in normal form and not a numeric value. There are two possibilities: either $\mathtt{g}_1$ is `true` or `false`, or else $\mathtt{g}_1$ itself is not a value, hence stuck. In the first case, rule E-SUCC-WRONG immediately yields $\mathtt{g} \xrightarrow{w} \mathtt{wrong}$; in the second case, the induction hypothesis gives us $\mathtt{g}_1 \xrightarrow{w}{}^* \mathtt{wrong}$ and we proceed as before.

Other cases:

Similar. □

Lemmas A.2 and A.3 together give us the "only if" ($\Longrightarrow$) half of Proposition A.1. For the other half, we need to show that a term that is "about to go wrong" in the augmented semantics is stuck in the original semantics.

A.4 Lemma: If $\mathtt{g} \xrightarrow{w} \mathtt{t}$ in the augmented semantics and `t` contains `wrong` as a subterm, then g is stuck in the original semantics. □

Proof: Straightforward induction on (augmented) evaluation derivations. □

Combining this with Lemma A.2 yields the "if" ($\Longleftarrow$) half of Proposition A.1, and we are finished.

Solution to 4.2.2: 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 4.2.5: Here's a simple one:

```
equal = λm. λn.
          and (iszro (m prd n))
              (iszro (n prd m));
```

Solution to 4.2.6: 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 4.2.7: Since call-by-value evaluation stops at abstractions, the result of `factorial` c_3 is actually a large term containing a good deal of "latent computation" that will only be performed later when the arguments `s` and `z` are supplied. Here is `factorial` c_2 (which is quite a bit smaller!):

```
factorial c2;
```

```
▶ (λs.
     λz.
       (λs'.
          λz'.
            (λs''. λz''. s'' z'') s'
            ((λs''. λz''.z'') s' z'))
       s
       ((λs'.
           λz'.
             (λs''.
                λz''.
                  (λs'''. λz'''. s''' z''') s''
                  ((λs'''. λz'''.z''') s'' z''))
             s'
             ((λs''. λz''.z'') s' z'))
        s
        z))
```

Solution to 4.2.8:

```
ff = λf. λl.
       test (isnull l)
          (λx. c₀) (λx. (plus (head l) (f (tail l)))) c₀;
sumlist = fixedpoint ff;

l = cons c₂ (cons c₃ (cons c₄ nil));
equal (sumlist l) c₉;
```

▸ (λx. λy. x)

Solution to 4.3.3: By induction on the size of t. (That is, assuming the desired property for all terms smaller than t, we must prove it for t itself; if we can do this, we may conclude that the property holds for all 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: $|FV(\mathtt{t})| = |FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2)| \leq |FV(\mathtt{t}_1)| + |FV(\mathtt{t}_2)| \leq size(\mathtt{t}_1) + size(\mathtt{t}_2) < size(\mathtt{t})$.

Solution to 5.1.13: If Γ is a naming context, write $\Gamma(\mathtt{x})$ for the index of x in Γ, counting from the right. Now, the property that we want is that

$$removenames_\Gamma(\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t}) \quad = \quad \{\Gamma(\mathtt{x}) \mapsto removenames_\Gamma(\mathtt{s})\}(removenames_\Gamma(\mathtt{t})).$$

The proof proceeds by induction on t, using Definitions 4.3.5 and 5.1.9, some simple calculations, and some easy lemmas about *removenames* and the other basic operations on terms. Convention 4.3.4 plays a crucial role in the abstraction case.

Solution to 6.3.8: Here's a counterexample: the term `(if false then true else 0)` is ill-typed, but evaluates to the well-typed term `0`.

Solution to 7.2.3: T-TRUE and T-FALSE are introduction rules. T-IF is an elimination rule.

Solution to 7.3.1: Because the set of type expressions is empty (there is no base case in the syntax of types).

Solution to 26.4.2: The proof proceeds by induction on a derivation of $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$. There are cases for each of the typing rules (or, equivalently, each of the possible forms of $\mathtt{t}$).

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

There are two sub-cases to consider, depending on whether $\mathtt{z}$ is the same as $\mathtt{x}$ 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: $\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T_2}.\mathtt{t_1}$
$\mathtt{T} = \mathtt{T_2}\to\mathtt{T_1}$
$\Gamma, \mathtt{x}{:}\mathtt{S}, \mathtt{y}{:}\mathtt{T_2} \vdash \mathtt{t_1} : \mathtt{T_1}$

First note that, by convention 4.3.4, we may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin FV(\mathtt{s})$. Using weakening and permutation on the given subderivation, we obtain $\Gamma, \mathtt{y}{:}\mathtt{T_2}, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_1} : \mathtt{T_1}$. By the induction hypothesis, $\Gamma, \mathtt{y}{:}\mathtt{T_2} \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} : \mathtt{T_1}$. By T-ABS, $\Gamma \vdash \lambda\mathtt{y}{:}\mathtt{T_2}.\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$. But this is the needed result, since, by the definition of substitution, $\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T_1}.\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1}$.

Case T-APP: $\mathtt{t} = \mathtt{t_1}\ \mathtt{t_2}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_2} : \mathtt{T_2}$
$\mathtt{T} = \mathtt{T_2}$

By the induction hypothesis, $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} : \mathtt{T_2}\to\mathtt{T_1}$ and $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_2} : \mathtt{T_2}$. By T-APP, $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1}\ \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_2} : \mathtt{T}$, i.e., $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}(\mathtt{t_1}\ \mathtt{t_2}) : \mathtt{T}$.

Case T-TRUE, T-FALSE: $\mathtt{t} = \mathtt{0}$
$\mathtt{T} = \mathtt{Bool}$

Then $\{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t} = \mathtt{0}$, and the desired result, $\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t} : \mathtt{T}$, is immediate.

Case T-IF: $\mathtt{t} = \mathtt{if}\ \mathtt{t_1}\ \mathtt{then}\ \mathtt{t_2}\ \mathtt{else}\ \mathtt{t_3}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_1} : \mathtt{Bool}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_2} : \mathtt{T}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t_3} : \mathtt{T}$

The induction hypothesis yields

$$\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_1} : \mathtt{Bool}$$
$$\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_2} : \mathtt{T}$$
$$\Gamma \vdash \{\mathtt{x} \mapsto \mathtt{s}\}\mathtt{t_3} : \mathtt{T},$$

from which the result follows by T-IF. □

Solution to 7.4.11: The term `(λx:Bool. λy:Bool. y) (true true)` is ill typed, but reduces to `(λy:Bool.y)`, which is well typed.

Solution to 9.4.1: Here are the most important additions:

```
type term =
    ...
  | TmLet of info * string * term * term

let tmmap onvar =
  let rec walk c = function
    ...
  | TmLet(fi,x,t1,t2) → TmLet(fi,x,walk c t1,walk (c+1) t2)
    ...

let rec eval1 ctx = function
    ...
  | TmLet(fi,x,v1,t2) when (isval ctx v1) →
      tmsubstsnip v1 t2
  | TmLet(fi,x,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmLet(fi, x, t1', t2)
    ...

let rec typeof ctx t =
  match t with
    ...
  | TmLet(fi,x,t1,t2) →
     let tyT1 = typeof ctx t1 in
     let ctx' = addbinding ctx x (VarBind(tyT1)) in
     (typeof ctx' t2)
```

For the sake of completeness, here are the additions to the lexing, parsing, and printing code. In the parser, we add two tokens

```
%token <Support.Error.info> LET
%token <Support.Error.info> IN
```

and one new production in the `Term` class:

```
Term :
    ...
  | LET LCID EQ Term IN Term
       fun ctx → TmLet($1, $2.v, $4 ctx, $6 (addname ctx $2.v))
  | LET USCORE EQ Term IN Term
       fun ctx → TmLet($1, "_", $4 ctx, $6 (addname ctx "_"))
```

In the lexer, we add two corresponding keyword definitions:

```
("let", fun i → Parser.LET i);
("in", fun i → Parser.IN i);
```

In the printing routines, we add one new clause:

```
let rec prtm_Term outer ctx = function
    ...
  | TmLet(fi, x, t1, t2) →
       obox0();
       pr "let "; pr x; pr " = ";
       prtm_Term false ctx t1;
       print_space();
       pr "in ";
       prtm_Term false (addname ctx x) t2;
       cbox()
```

Solution to 9.6.1:

Typed record patterns $\rightarrow$ {} let *pat* **(typed)**

Pattern typing rules

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

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

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

Solution to 11.2: Let μ be a store with a single location $\mathtt{l}$

$$\mu \quad = \quad \mathtt{l} = \lambda\mathtt{x{:}Unit.\ (!l)(x)},$$

and Γ the empty context. Then μ is well typed with respect to both of the following store typings:

$$\begin{aligned} \Sigma_1 &= \mathtt{l{:}Unit{\to}Unit} \\ \Sigma_2 &= \mathtt{l{:}Unit{\to}(Unit{\to}Unit)}. \end{aligned}$$

Solution to 11.6: There are well-typed terms in this system that are not strongly normalizing. For example, consider the following:

```
t1 = λr:Ref (Unit→Unit).
       (r:=(λx:Unit. (!r)x);
        (!r) unit);
t2 = ref (λx:Unit. x);
```

Applying `t1` to `t2` yields a (well-typed) divergent term.

Solution to 12.1.2: `Top` and `{}` are similar in that they are both, in some sense, maximal elements. The difference is that `{}` is a supertype of every record type, but only record types, while `Top` is above every type, including `{}`.

Solution to 12.1.3:

1. I count six:

```
{a:Top,b:Top}
{b:Top,a:Top}
{a:Top}
{b:Top}
{}
Top.
```

2. For example, let

$$
\begin{array}{lcl}
S_0 & = & \{\} \\
S_1 & = & \{a:Top\} \\
S_2 & = & \{a:Top,b:Top\} \\
S_2 & = & \{a:Top,b:Top,c:Top\} \\
\text{etc.} & &
\end{array}
$$

3. For example, let

$$
\begin{array}{lcl}
T_0 & = & S_0 \to Top \\
T_1 & = & S_1 \to Top \\
T_2 & = & S_2 \to Top \\
\text{etc.} & &
\end{array}
$$

Solution to 12.1.4:

1. No. If there were, it would have to be either an arrow or a record (it obviously can't be `Top`). But a record type would not be a subtype of any of the arrow types, and vice versa.

2. No. If there *were* such an arrow type $T_1 \to T_2$, its domain type T_1 would have to be a subtype of every other type S_1; but we have just seen that this is not possible.

Solution to 12.1.5: Adding this rule would not be a good idea if we want to keep the existing untyped evaluation semantics. The new rule would allow us to derive, for example, that `Nat`$\times$`Bool` <: `Nat`, which would mean that the stuck term `(succ (5,true))` would be well typed, violating the progress theorem. The rule does make sense in a "coercion semantics" (see Section **??**), though even there it raises some algorithmic difficulties for subtype checking.

Solution to 17.3.9: Here are the algorithmic constraint generation rules:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash_F \mathtt{x} : \mathtt{T} \,|_F\, \{\}} \qquad \text{(CT-VAR)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{S} \vdash_F \mathtt{t}_1 : \mathtt{T} \,|_{F'}\, C \qquad \mathtt{x} \notin \mathit{dom}(\Gamma)}{\Gamma \vdash_F \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t}_1 : \mathtt{S} \to \mathtt{T} \,|_{F'}\, C} \qquad \text{(CT-ABS)}$$

$$\frac{\Gamma \vdash_F \mathtt{t}_1 : \mathtt{T}_1 \,|_{F'}\, C_1 \qquad \Gamma \vdash_{F'} \mathtt{t}_2 : \mathtt{T}_2 \,|_{F''}\, C_2 \qquad F'' = \mathtt{X}, F'''}{\Gamma \vdash_F \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{X} \,|_{F'''}\, C_1 \cup C_2 \cup \{\mathtt{T}_1 = \mathtt{T}_2 \to \mathtt{X}\}} \qquad \text{(CT-APP)}$$

$$\Gamma \vdash_F \mathtt{0} : \mathtt{Nat} \,|_F\, \{\} \qquad \text{(CT-ZERO)}$$

$$\frac{\Gamma \vdash_F \mathtt{t}_1 : \mathtt{T} \,|_{F'}\, C}{\Gamma \vdash_F \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat} \,|_{F'}\, C \cup \{\mathtt{T} = \mathtt{Nat}\}} \qquad \text{(CT-SUCC)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_F \mathtt{t}_1 : \mathtt{T}_1 \,|_{F'}\, C_1 \qquad \Gamma \vdash_{F'} \mathtt{t}_2 : \mathtt{T}_2 \,|_{F''}\, C_2 \qquad \Gamma \vdash_{F''} \mathtt{t}_3 : \mathtt{T}_3 \,|_{F'''}\, C_3 \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\mathtt{T}_1 = \mathtt{Nat}, \mathtt{T}_2 = \mathtt{T}_3 \to \mathtt{T}_3\}\end{array}}{\Gamma \vdash_F \mathtt{iter}\ \mathtt{T}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3 : \mathtt{T}_3 \,|_{F'''}\, C'} \qquad \text{(CT-ITER)}$$

The equivalence of the original rules and the algorithmic presentation can be stated as follows:

1. (Soundness) If $\Gamma \vdash_F \mathtt{t} : \mathtt{T} \,|_{F'}\, C$ and the variables mentioned in Γ and `t` do not appear in F, then $\Gamma \vdash \mathtt{t} : \mathtt{T} \,|_{F' \setminus F}\, C$.

2. (Completeness) If $\Gamma \vdash \mathtt{t} : \mathtt{T} \,|_{\mathcal{X}}\, C$, then there is some permutation F of the names in $\mathcal{X}$ such that $\Gamma \vdash_F \mathtt{t} : \mathtt{T} \,|_{\emptyset}\, C$.

Both parts are proved by straightforward induction on derivations. For the application case in part 1, the following lemma is useful:

> If the type variables mentioned in Γ and $\mathtt{t}$ do not appear in F and if $\Gamma \vdash_{\mathsf{F}} \mathtt{t} : \mathtt{T} \,|_{\mathsf{F}'}\, \mathsf{C}$, then the type variables mentioned in $\mathtt{T}$ and C do not appear in $\mathsf{F}' \setminus \mathsf{F}$.

For the same case in part 2, the following lemma is used:

> If $\Gamma \vdash_{\mathsf{F}} \mathtt{t} : \mathtt{T} \,|_{\mathsf{F}'}\, \mathsf{C}$, then $\Gamma \vdash_{\mathsf{F},\mathsf{G}} \mathtt{t} : \mathtt{T} \,|_{\mathsf{F}',\mathsf{G}}\, \mathsf{C}$, where G is any sequence of fresh variable names.

Solution to 17.3.10: Representing constraint sets as lists of pairs of types, the constraint generation algorithm is a direct transcription of the inference rules given above.

```
let rec recon ctx nextuvar t =
  match t with
      TmVar(fi,i,_) →
        let tyS = gettype fi ctx i in
        (tyS, nextuvar, [])
    | TmAbs(fi, x, tyS, t1) →
        let ctx' = addbinding ctx x (VarBind(tyS)) in
        let (tyT1,nextuvar1,constr1) = recon ctx' nextuvar t1 in
        (TyArr(tyS, tyT1), nextuvar1, constr1)
    | TmApp(fi,t1,t2) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let NextUVar(tyX,nextuvar') = nextuvar2() in
        let newconstr = [(tyT1,TyArr(tyT2,TyId(tyX)))] in
        ((TyId(tyX)), nextuvar', (newconstr  constr1  constr2))
    | TmZero(fi) → (TyNat, nextuvar, [])
    | TmSucc(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, [(tyT1,TyNat)])
    | TmPred(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, [(tyT1,TyNat)])
    | TmIsZero(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyBool, nextuvar1, [(tyT1,TyNat)])
    | TmTrue(fi) → (TyBool, nextuvar, [])
    | TmFalse(fi) → (TyBool, nextuvar, [])
    | TmIf(fi,t1,t2,t3) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let (tyT3,nextuvar3,constr3) = recon ctx nextuvar2 t3 in
        let newconstr = [(tyT1,TyBool); (tyT2,tyT3)] in
        (tyT3, nextuvar3, List.concat [newconstr; constr1; constr2; constr3])
```

Solution to 17.4.4:

$\{$X = Nat, Y = X→X$\}$	$\{$X ↦ Nat, Y ↦ Nat→Nat$\}$
$\{$Nat→Nat = X→Y$\}$	$\{$X ↦ Nat, Y ↦ Nat$\}$
$\{$X→Y = Y→Z, Z = U→W$\}$	$\{$X ↦ U→W, Y ↦ U→W, Z ↦ U→W$\}$
$\{$Nat = Nat→Y$\}$	Not unifiable
$\{$Y = Nat→Y$\}$	Not unifiable

Solution to 17.4.7: One more variant of substitution is needed in the unification function—the application of a substitution to all the types in some constraint set:

```
let substinconstr tyX tyT constr =
  List.map
    (fun (tyS1,tyS2) →
       (substinty tyX tyT tyS1, substinty tyX tyT tyS2))
    constr
```

Also crucial is the "occur-check" that detects circular dependencies:

```
let occursin tyX tyT =
  let rec o = function
      TyArr(tyT1,tyT2) → o tyT1 || o tyT2
    | TyNat → false
    | TyBool → false
    | TyId(s) → (s=tyX)
  in o tyT
```

The unification function is now a direct transcription of the rules given on page 148. As usual, it takes a file position and string as extra arguments to be used in printing error messages when unification fails. (This pedagogical version of the unifier does not work very hard to print useful error messages. In practice, "explaining" type errors can be one of the hardest parts of engineering a production compiler for a language with type reconstruction.)

```
let unify fi ctx msg constr =
  let rec u constr =
    match constr with
      [] →
        []
    | (tyS,TyId(tyX)) :: rest →
        if tyS = TyId(tyX) then
          u rest
        else if occursin tyX tyS then
          error fi (msg ^ ": circular constraints")
        else
          List.append (u (substinconstr tyX tyS rest)) [(TyId(tyX),tyS)]
    | (TyId(tyX),tyT) :: rest →
        if tyT = TyId(tyX) then
```

```
        u rest
      else if occursin tyX tyT then
        error fi (msg ^ ": circular constraints")
      else
        List.append (u (substinconstr tyX tyT rest)) [(TyId(tyX),tyT)]
  | (TyNat,TyNat) :: rest →
      u rest
  | (TyBool,TyBool) :: rest →
      u rest
  | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) :: rest →
      u ((tyS1,tyT1) :: (tyS2,tyT2) :: rest)
  | (tyS,tyT)::rest →
      error fi "Unsolvable constraints"
in
  u constr
```

Solution to 17.5.6: Extending the type reconstruction algorithm to handle records is not straightforward, though it can be done. The main difficulty is that it is not clear what constraints should be generated for a record projection. A naive first attempt would be

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C}{\Gamma \vdash \mathtt{t}.\mathtt{l}_i : \mathtt{X} \mid_{\mathcal{X} \cup \{\mathtt{X}\}} C \cup \{\mathtt{T} = \{\!|\mathtt{l}_i\!:\!\mathtt{X}|\!\}\}}$$

but this is not satisfactory, since this rule says, in effect, that the field $\mathtt{l}_i$ can only be projected from a record containing *just* the field $\mathtt{l}_i$ and no others.

An elegant solution was proposed by Wand [Wan87] and further developed by Wand [Wan88, Wan89], Remy [Rém89, Rém90], and others. We introduce a new kind of unification variable, called a **row variable**, ranging not over types but over "rows" of field labels and associated types. Using row variables, the constraint generation rule for field projection can be written

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{T} \mid_{\mathcal{X}} C}{\Gamma \vdash \mathtt{t}_0.\mathtt{l}_i : \mathtt{X} \mid_{\mathcal{X} \cup \{\mathtt{X},\sigma,\rho\}} C \cup \{\mathtt{T} = \{\!|\rho|\!\},\ \rho = \mathtt{l}_i\!:\!\mathtt{X} \oplus \sigma\}} \qquad \text{(CT-PROJ)}$$

where σ and ρ are row variables and the operator $\oplus$ combines two rows (assuming that their fields are disjoint). That is, the term $\mathtt{t}.\mathtt{l}_i$ has type $\mathtt{X}$ if $\mathtt{t}$ has a record type with fields ρ, where ρ contains the field $\mathtt{l}_i\!:\!\mathtt{X}$ and some other fields σ.

The constraints generated by this refined algorithm are more complicated than the simple sets of equations between types with unification variables of the original reconstruction algorithm, since the new constraint sets also involve the associative and commutative operator $\oplus$. A simple form of **equational unification** is needed to find solutions to such constraint sets.

Solution to 15.3.1: Lists:
Hungry functions:

```
f =
  fix
    (λf: Nat→Hungry.
     λn:Nat.
       fold [Hungry] f);
ff = fold [Hungry] f;
```

```
▶ Hungry = μA. Nat → A: *
  f : Nat → Hungry
  ff : Hungry
```

```
ff1 = (unfold [Hungry] ff) 0;
```

```
▶ ff1 : Hungry
```

```
ff2 = (unfold [Hungry] ff1) 2;
```

```
▶ ff2 : Hungry
```

Fixed points:

```
fixedpoint_T =
  λf:T→T.
    (λx:(μA.A→T). f ((unfold [μA.A→T] x) x))
    (fold [μA.A→T] (λx:(μA.A→T). f ((unfold [μA.A→T] x) x)));
diverge_T = λX. fixedpoint_T (λx:T. x);
```

Untyped lambda-calculus:

```
D = μX. X→X;
lam = λf:D→D. fold [D] f;
ap = λf:D. λa:D. (unfold [D] f) a;
```

Untyped lambda-calculus with numbers and booleans:
Objects:

```
p =
  let create =
    fix
      (λcr: {x:Nat}→Counter.
        λs: {x:Nat}.
          fold [Counter]
            {get = s.x,
             inc = λ_:Unit. cr {x=succ(s.x)}})
  in
    create {x=0};
p1 = (unfold [Counter] p).inc unit;
(unfold [Counter] p1).get;
```

```
▸ T = Nat: *
  fixedpoint_T : (T→T) → T
  diverge_T : ∀X. T
  D = μX. X → X: *
  lam : (D→D) → D
  ap : D → D → D
  Counter = μP. {get:Nat, inc:Unit→P}: *
  p : Counter
  p1 : Counter
  ((unfold [Counter] p1).get) : Nat
```

Solution to 21.2.3: Just one additional rule is needed:

$$\frac{\begin{array}{c}\Gamma \mathrel{|\!\blacktriangleright} \mathtt{b} : \mathtt{B} \qquad \Gamma \vdash \mathtt{B} \Uparrow \mathtt{Bool} \\ \Gamma \mathrel{|\!\blacktriangleright} \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \mathrel{|\!\blacktriangleright} \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_1 \vee \mathtt{T}_2 = \mathtt{J}\end{array}}{\Gamma \mathrel{|\!\blacktriangleright} \mathtt{if\ b\ then\ t}_1\ \mathtt{else\ t}_2 : \mathtt{J}} \qquad \text{(TA-COND)}$$

where the final premise says that J is a join of $\mathtt{T}_1$ and $\mathtt{T}_2$ in Γ.

Solution to 21.4.2: We begin by giving a pair of algorithms that, when presented with Γ, S, and T, calculate a pair of types J and M, which we will claim are a join and meet, respectively, of S and T. (The second algorithm may also fail, in which case we claim that S and T have no meet in Γ.)

We write $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ for "M is the meet of S and T in context Γ" and $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$ for "J is the join of S and T in Γ." The algorithms are defined simultaneously as follows. (Note that some of the cases of the definition overlap. Since it is technically convenient to treat meet and join as functions rather than relations, we stipulate that the first clause that applies must be chosen.)

$$
\Gamma \vdash \mathtt{S} \wedge \mathtt{T} \quad = \quad \begin{cases}
\mathtt{S} & \text{if } \Gamma \vdash \mathtt{S} <: \mathtt{T} \\
\mathtt{T} & \text{if } \Gamma \vdash \mathtt{T} <: \mathtt{S} \\
\mathtt{J}{\to}\mathtt{M} & \text{if } \mathtt{S} = \mathtt{S}_1{\to}\mathtt{S}_2 \\
 & \quad \mathtt{T} = \mathtt{T}_1{\to}\mathtt{T}_2 \\
 & \quad \Gamma \vdash \mathtt{S}_1 \vee \mathtt{T}_1 = \mathtt{J} \\
 & \quad \Gamma \vdash \mathtt{S}_2 \wedge \mathtt{T}_2 = \mathtt{M} \\
\forall\mathtt{X}<:\mathtt{U}.\mathtt{M} & \text{if } \mathtt{S} = \forall\mathtt{X}<:\mathtt{U}.\mathtt{S}_2 \\
 & \quad \mathtt{T} = \forall\mathtt{X}<:\mathtt{U}.\mathtt{T}_2 \\
 & \quad \Gamma, \mathtt{X}<:\mathtt{U} \vdash \mathtt{S}_2 \wedge \mathtt{T}_2 = \mathtt{M} \\
\textit{fail} & \text{otherwise}
\end{cases}
$$

$$
\Gamma \vdash \mathtt{S} \vee \mathtt{T} \quad = \quad \begin{cases}
\mathtt{T} & \text{if } \Gamma \vdash \mathtt{S} <: \mathtt{T} \\
\mathtt{S} & \text{if } \Gamma \vdash \mathtt{T} <: \mathtt{S} \\
\mathtt{J} & \text{if } \mathtt{S} = \mathtt{X} \text{ with } \mathtt{X}<:\mathtt{U} \in \Gamma \text{ and } \Gamma \vdash \mathtt{U} \vee \mathtt{T} = \mathtt{J} \\
\mathtt{J} & \text{if } \mathtt{T} = \mathtt{X} \text{ with } \mathtt{X}<:\mathtt{U} \in \Gamma \text{ and } \Gamma \vdash \mathtt{S} \vee \mathtt{U} = \mathtt{J} \\
\mathtt{M}{\to}\mathtt{J} & \text{if } \mathtt{S} = \mathtt{S}_1{\to}\mathtt{S}_2 \\
 & \quad \mathtt{T} = \mathtt{T}_1{\to}\mathtt{T}_2 \\
 & \quad \Gamma \vdash \mathtt{S}_1 \wedge \mathtt{T}_1 = \mathtt{M} \\
 & \quad \Gamma \vdash \mathtt{S}_2 \vee \mathtt{T}_2 = \mathtt{J} \\
\forall\mathtt{X}<:\mathtt{U}.\mathtt{J} & \text{if } \mathtt{S} = \forall\mathtt{X}<:\mathtt{U}.\mathtt{S}_2 \\
 & \quad \mathtt{T} = \forall\mathtt{X}<:\mathtt{U}.\mathtt{T}_2 \\
 & \quad \Gamma, \mathtt{X}<:\mathtt{U} \vdash \mathtt{S}_2 \vee \mathtt{T}_2 = \mathtt{J} \\
\mathtt{Top} & \text{otherwise}
\end{cases}
$$

It is easy to check that $\wedge$ and $\vee$ are total functions: for every Γ, $\mathtt{S}$, and $\mathtt{T}$, there are unique types $\mathtt{M}$ and $\mathtt{J}$ such that $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ and $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$: just note that the total weight of $\mathtt{S}$ and $\mathtt{T}$ with respect to Γ is always reduced in recursive calls.

Now let us verify that these definitions do indeed calculate meets and joins in the subtype relation. The argument into two parts: Proposition A.5 shows that the calculated meet is a lower bound of $\mathtt{S}$ and $\mathtt{T}$ and the join is an upper bound; Proposition A.6 then shows that the calculated meet is greater than every common lower bound of $\mathtt{S}$ and $\mathtt{T}$ and the join is less than every common upper bound.

A.5 Proposition:

1. If $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$, then $\Gamma \vdash \mathtt{M} <: \mathtt{S}$ and $\Gamma \vdash \mathtt{M} <: \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$, then $\Gamma \vdash \mathtt{S} <: \mathtt{J}$ and $\Gamma \vdash \mathtt{T} <: \mathtt{J}$. □

Proof: By a straightforward induction on the size of a "derivation" of $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ or $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$ (i.e., the number of recursive calls to the definitions of $\wedge$ and $\vee$ needed to calculate $\mathtt{M}$ or $\mathtt{J}$). □

A.6 Proposition:

1. Suppose that $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ and, for some $\mathtt{L}$, that $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{S}$ and $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{T}$. Then $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{M}$.

2. Suppose that $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$ and, for some $\mathtt{U}$, that $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{U}$ and $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{U}$. Then $\Gamma \vdash \mathtt{J} \mathrel{<:} \mathtt{U}$. □

Proof: Simultaneously, by induction on the total size of derivations of $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{S}$ and $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{T}$ (for part 1) or $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{U}$ and $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{U}$ (for part 2).

In both parts, let us deal first with the case where $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ or $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{S}$. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$, then $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{S}$ and $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{T}$. But then $\Gamma \vdash \mathtt{L} \mathrel{<:} \mathtt{M}$ and $\Gamma \vdash \mathtt{J} \mathrel{<:} \mathtt{U}$ by assumption. Similarly when $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{S}$.

To complete the proofs, assume that $\Gamma \vdash \mathtt{S} \not\mathrel{<:} \mathtt{T}$ and $\Gamma \vdash \mathtt{T} \not\mathrel{<:} \mathtt{S}$ and consider the two parts of the proposition in turn.

1. Consider the form of $\mathtt{L}$.

   Case: $\mathtt{L} = \mathtt{Top}$

   Then $\mathtt{S} = \mathtt{Top}$ and $\mathtt{T} = \mathtt{Top}$, so $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ and this case has already been dealt with.

   Case: $\mathtt{L} = \mathtt{X}$

   If either $\mathtt{S}$ or $\mathtt{T}$ is the variable $\mathtt{X}$, then we have either $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ or $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{S}$; these cases have already been dealt with.

   Otherwise, by the inversion lemma, we have $\Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{S}$ and $\Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{T}$, where $\mathtt{X}\mathrel{<:}\mathtt{U} \in \Gamma$. The induction hypothesis yields $\Gamma \vdash \mathtt{U} \mathrel{<:} \mathtt{M}$, from which $\Gamma \vdash \mathtt{X} \mathrel{<:} \mathtt{M}$ follows by S-TVAR and S-TRANS.

   Case: $\mathtt{L} = \mathtt{A}\to\mathtt{B}$

   First note that, since $\Gamma \vdash \mathtt{S} \not\mathrel{<:} \mathtt{T}$ and $\Gamma \vdash \mathtt{T} \not\mathrel{<:} \mathtt{S}$, neither $\mathtt{S}$ nor $\mathtt{T}$ can be $\mathtt{Top}$, so the only remaining case is where

   $$\begin{aligned} &\mathtt{S} = \mathtt{V}\to\mathtt{P} \\ &\mathtt{T} = \mathtt{W}\to\mathtt{Q} \end{aligned}$$

   with

   $$\begin{aligned} &\Gamma \vdash \mathtt{V} \mathrel{<:} \mathtt{A} \\ &\Gamma \vdash \mathtt{B} \mathrel{<:} \mathtt{P} \\ &\Gamma \vdash \mathtt{W} \mathrel{<:} \mathtt{A} \\ &\Gamma \vdash \mathtt{B} \mathrel{<:} \mathtt{Q}. \end{aligned}$$

   Also, by the definition of meets, $\mathtt{M} = \mathtt{J}\to\mathtt{N}$ with

   $$\begin{aligned} &\Gamma \vdash \mathtt{V} \vee \mathtt{W} = \mathtt{J} \\ &\Gamma \vdash \mathtt{P} \wedge \mathtt{Q} = \mathtt{N}. \end{aligned}$$

Applying the induction hypothesis, we obtain

$$\Gamma \vdash \mathtt{J} <: \mathtt{A}$$
$$\Gamma \vdash \mathtt{B} <: \mathtt{N},$$

from which $\Gamma \vdash \mathtt{L} <: \mathtt{M}$ follows by S-ARROW.

Case: $\mathtt{L} = \forall \mathtt{X}{<:}\mathtt{U}.\mathtt{B}$

Similar.

2. First, observe that, by the inversion lemma, there are two possibilities for S—either it is a variable whose upper bound is a subtype of U, or else its form depends on the form of U—and similarly for T. Let us deal first with the first possibility for both S and T, leaving the structural cases to be considered in detail.

   If $\mathtt{S} = \mathtt{Y}$ with $\Gamma(\mathtt{Y}) = \mathtt{R}$ and $\Gamma \vdash \mathtt{R} <: \mathtt{U}$, then, by the definition of $\vee$, we have $\Gamma \vdash \mathtt{R} \vee \mathtt{T} = \mathtt{J}$. The induction hypothesis now yields $\Gamma \vdash \mathtt{J} <: \mathtt{U}$, and we are finished. Similarly if T is a variable bounded by U.

   Now suppose that the forms of S and T depend on the form of U.

   Case: $\mathtt{U} = \mathtt{Top}$

   Immediate.

   Case: $\mathtt{U} = \mathtt{X}$

   By the inversion lemma, we have $\mathtt{S} = \mathtt{X}$ and $\mathtt{T} = \mathtt{X}$; we have dealt with this case already.

   Case: $\mathtt{U} = \mathtt{A}{\to}\mathtt{B}$

   Then by the inversion lemma we have

   $$\mathtt{S} = \mathtt{V}{\to}\mathtt{P}$$
   $$\mathtt{T} = \mathtt{W}{\to}\mathtt{Q}$$

   with

   $$\Gamma \vdash \mathtt{A} <: \mathtt{V}$$
   $$\Gamma \vdash \mathtt{P} <: \mathtt{B}$$
   $$\Gamma \vdash \mathtt{A} <: \mathtt{W}$$
   $$\Gamma \vdash \mathtt{Q} <: \mathtt{B}.$$

   The rest of the argument proceeds as in the corresponding case in part 1.

   Case: $\mathtt{U} = \forall \mathtt{X}{<:}\mathtt{A}.\mathtt{B}$

   Similar. □

Solution to 21.4.3:

1. Using the inversion lemma, I count 9 common subtypes of S and T:

```
∀X<:A′→B. A→B′
∀X<:A′→B. Top→B′
∀X<:A′→B. X
∀X<:A′→Top. A→B′
∀X<:A′→Top. Top→B′
∀X<:A′→Top. X
∀X<:Top. A→B′
∀X<:Top. Top→B′
∀X<:Top. X.
```

2. Both

```
∀X<:A′→B. A→B′
```

and

```
∀X<:A′→B. X
```

are lower bounds for S and T, but these two types have no common supertype that is also a subtype of S and T.

3. Consider S→Top and T→Top. (Or, if you like, ∀X<:A′→B. A→B′ and ∀X<:A′→B. X.)

Solution to ??: Note that the two kinds of quantifiers—bounded and unbounded—should not be allowed to mix: there should be a subtyping rule for comparing two bounded quantifiers and another for two unbounded quantifiers, but no rule for comparing a bounded to an unbounded quantifier. Otherwise we'd be right back where we started!

1. See [KS92] for details.

2. No. In any practical language with subtyping, we will want to allow width subtyping on record types. But the empty record type is a kind of maximal type (among record types), and it can be used to cause divergence in the subtype checker using a modified version of Ghelli's example. If

```
T = ∀[X<:{||}|] ¬{|a: ∀[Y<:X|]¬Y|}
```

then the input

$$X_0\texttt{<:}\{\!|\texttt{a:T}|\!\} \vdash X_0 <: \{\!|\texttt{a: }\forall[X_1\texttt{<:}X_0|]\neg X_1|\!\}$$

will cause the subtype checker to diverge. [Martin Hofmann helped work out this example.]

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
nv	nv	numeric values
M, N, P, Q, 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
K, L	kK, kL	kinds
Γ, Δ	ctx	contexts
	fi	file position information
$\mathcal{J}$		arbitrary typing statements

B.2 Rule Naming Conventions

Prefix	Usage
E-	evaluation
XA-	exposure
K-	kinding
M-	matching
P-	pattern typing
Q-	type equivalence
QA-	algorithmic type equivalence

PREFIX	USAGE
RA-	algorithmic type reduction
S-	subtyping
SA-	algorithmic subtyping
T-	typing
TA-	algorithmic typing

Appendix C

Suggestions for Larger Projects

These need to be cleaned up, and I'd like to add several more.

This appendix collects some ideas for larger course projects based on the material in these notes and on the research literature on type systems. Most of the projects involve some additional reading and some implementation work.

The suggestions are roughly categorized into "T" [theoretical] projects, which are more open-ended and may require more reading, thinking, and design before getting started, and "I" [implementation] projects, which are somewhat more specific and involve getting down to implementation fairly early. This does not mean that the "T" projects are better – or even harder, ultimately – than "I" projects, but the "I" projects may be easier to get started on.

One note on the implementation-oriented projects. You should feel free to use any programming language you're comfortable with for building projects in... but if your favorite language happens to be C or Pascal, I urge you to think seriously about learning and using some higher-level language with built-in garbage collection and facilities for high-level symbol manipulation—for example, ML, Haskell, Scheme, Modula 3, or even Java.[1] All of these projects *can* be done in Pascal, C, or C++, but you'll spend more time than you can imagine chasing pointer bugs.

C.1 Objects

1. [I] In their book, *A Theory of Objects* [AC96] (and in a series of earlier articles [AC94b, AC94a, etc.]), Abadi and Cardelli have proposed a primitive calculus analogous to the lambda-calculus, but with objects (rather than functions) as the basic terms and message passing (rather than application) as the

[1] An object-oriented language like Java will probably be somewhat less convenient than a language with support for datatypes and pattern matching; if you do choose an OO language, you may want to structure your code according to the "visitor pattern" [GHJV94, FF98, etc.].

basic mechanism for computation. They develop several type systems for their object-calculus (OC). Implement one or more of these.

2. [T] Bruce, Cardelli, and Pierce wrote a paper [BCP99] comparing four different lambda-calculus encodings of objects—the "existential encoding" presented in Chapters 30 and **??**, the "recursive record" encoding mentioned in Chapter 15, and two other, hybrid, models with more refined properties. A number of different points of comparison are addressed in the paper, but different encodings of *classes* in the four object models are not considered.
   (a) Using the typechecker that we've used for exercises and for the examples in the notes, implement the examples in [BCP99].
   (b) Extend these examples to include encodings of classes and inheritance, following Chapter 30 in the case of the existential encoding and other papers (see the bibliography of [BCP99]) for the other encodings.
   (c) Compare and contrast.
   (d) If time remains, it is also interesting to compare and contrast the *imperative* variants of these four encodings—i.e., versions of the encodings where the instance variables of objects can be mutable `Ref` cells.

3. [I] Chapter **??** shows how to do some simple examples of object-oriented programming in $F_{<:}$. Using the `everything` typechecker from the course web directory, develop a more significant object-oriented program following the same lines.
   (a) To get started, implement a group of *collection classes* (for example, using the collection classes of Smalltalk [GR83] as a model). Some hints for how to do this can be found in [PT94].
   (b) Use these classes as a library to implement a larger object-oriented program of your choice.

C.2 Encodings of Logics

1. [I] The *Logical Framework* of Harper, Honsell, and Plotkin [HHP92] is a typed lambda-calculus intended as a meta-language for defining logics and proving theorems in them.
   (a) Implement LF
   (b) Use your implementation to encode a simple propositional logic and prove some small theorems.

2. [T] A generalization of the LF logic has been used as the core of Frank Pfenning's logic programming language ELF [Pfe89, Pfe91, Pfe94, Pfe96].

(a) Download and install the ELF system.

(b) Use it to encode the syntax and typing rules of the simply typed lambda-calculus with subtyping and prove some simple theorems about the system, along the lines of [MP91].

C.3 Type Inference

1. [T] Implement a type inference system for the simply typed lambda-calculus (or ML) with subtyping. (The literature on approaches to this problem is huge. See [Pot97, Pot98], for example.)

2. [I] *Row variables*, proposed by Mitch Wand [Wan87, Wan88, Wan89] and subsequently refined by Didier Remy [Rém89, Rém90, RV97], provide an alternative to subtyping as a foundation for object-oriented programming languages, and can coexist smoothly with ML-style type inference. Implement a type system with type inference and row variables.

C.4 Other Type Systems

1. [T] Implement a lambda-calculus with intersection types [Rey88, Pie97] (or, for a bit more challenge, with both intersection and union types [Pie91b, Pie90, BDCd95, Hay91]).

2. [T] Implement a *linear* type system for the lambda-calculus [Wad91, Wad90, TWM95].

3. [I] Implement a *typed assembly language* in the style of Morrisett and co-workers [MWCG98, MCGW98, MCG$^+$99].

4. [T] A great variety of type systems for *extensible records*, developed by various researchers, are summarized and unified in Cardelli and Mitchell's encyclopedic article, *Operations on Records* [CM91]. Implement a simple variant of their system.

5. [I] *Effect type systems* [JG91, TJ92], which track the computational effects of functions (memory reads and writes, allocation, etc.) in addition to their input-output behavior, have been used for a variety of purposes. Perhaps most surprisingly, Tofte and his co-workers have used "region inference" techniques to build an ML compiler that runs without a garbage collector [TT97, TB98]. Implement a simple effect inference algorithm.

6. [T] Harper and Lillibridge [HL94] and Leroy [Ler96] have independently proposed similar accounts of *modules* (in the style of ML) using the type-theoretic tools of existential types and (a limited form of) dependent types.

These accounts place powerful module systems like those of Standard ML [MTH90] and Objective Caml [Ler95] on a well-understood and tractable theoretical foundation. Read and implement one of these papers.

7. [I] A series of papers [ACPP91, LM91, ACPR95] have proposed adding a type `Dynamic` to statically typed languages, in order to provide a smooth interface between statically typed and dynamically typed data.

   (a) Add `Dynamic` to the implementation of the simply typed lambda-calculus, following [ACPP91].

   (b) Extend this implementation to System F (or even Fomega), using ideas in [ACPR95] and your own creativity.

C.5 Sources for Additional Ideas

The research literature is full of descriptions of interesting type systems and applications of type systems. The premier conference in the area is *Principles of Programming Languages (POPL)*. Other conferences with a high density of papers on type systems include *International Conference on Functional Programming (ICFP)*, *Typed Lambda Calculi and Applications (TLCA)*, *Theoretical Aspects of Computer Software (TACS)*, *Logic in Computer Science (LICS)*, (among many others). Leafing through a recent proceedings of any of these conferences should yield several papers involving type systems. Pick one, read it carefully, and build a simple implementation of the system it describes.

Appendix D

Bluffers Guide to OCaml

The systems discussed in the text have all been implemented in the Objective CAML dialect of ML. An excellent compiler for this language is freely available from INRIA.

Students who want to undertake the implementation exercises suggested in many chapters will need to pick up the basics of OCaml programming, if they are not already familiar with it. For programmers already familiar with another dialect of ML (e.g. Standard ML), the brief, but excellent, tutorial in the OCaml reference manual will do fine for this. Readers with no prior familiarity with any ML may wish to read some of Cousineau and Mauny's textbook [CM98].

For the convenience of readers who just want to understand enough of the programming examples presented here to translate them into another programming language, this appendix summarizes the key features used.

To be written. I have in mind something along the lines of the "Bluffers Guide to ML" in Baader and Nipkow's recent book, Term Rewriting and All That. *In fact, it might be possible to reprint that appendix entirely from this book, crediting the original authors and just translating,* mutatis mutandi, *SML to OCaml. (I have not talked to Baader and Nipkow about this.)*

Appendix E

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.

E.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

E.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`
$\to$	`->`
$'$a	`'a`
$'$b	`'b`
$'$c	`'c`
$'$d	`'d`

E.3 Running the Checker

```
Here's what you need to do to run it:
 1) make sure you've got ocaml installed
    1a) If you're running on a CIS system:
          add /pkg/ocaml-2.02/bin to your search path
          do 'ocamlc -v' to make sure it's there
    1b) If you are not running on a CIS system, you'll need to install the
        OCaml compiler, available from http://caml.inria.fr.  Please
        let me know if you need help with this.
 2) Grab the sources from
     http://www.cis.upenn.edu/~bcpierce/courses/700/checkers/fulluntyped.tar.gz
    or
     /mnt/saul/home/bcpierce/pub/courses/700/checkers/fulluntyped.tar.gz
 3) now do
      gunzip fulluntyped.tar.gz
      tar xvf fulluntyped.tar
      cd fulluntyped
    3a) if you're running on a Unix system, do
         make
        to create an executable file f
    3b) if you're running on a Windows system, do
         make windows
        to create an executable file f.exe
 4) create an input file containing one or more lambda-terms
 5) evaluate these terms by doing
      ./f <filename>
    where <filename> is the name of your file

In general, your input file can contain a sequence of commands, where
each command is either:

     t;        evaluate the term t and print its normal form
     x = t;    bind the name x to the term t

The terms in this file should look exactly like the examples in the
chapter, except that the character lambda is spelled out in full, e.g.:

         id = lambda x. x;
```

E.4 Old Instructions

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).

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.

Objective Caml

On CIS machines, you should be able to use the Objective CAML compiler simply by adding

```
/pkg/ocaml-XXX/bin
```

to your search path, where XXX is the latest version number.

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 `http://caml.inria.fr/lang`. It's very easy to install.

Bibliography

[AC94a] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *European Symposium on Programming (ESOP), Edinburgh, Scotland*, 1994.

[AC94b] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1994.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991. Preliminary version in Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (Austin, TX), January, 1989.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Preliminary version in Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, June 1992.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[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.

[Bar92a] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992.

[Bar92b] Henk Barendregt. Lambda calculi with types. In Gabbay Abramsky and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 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).

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1999. To appear in a special issue with papers from *Theoretical Aspects of Computer Software (TACS)*, September, 1997. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, June 1995.

[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.

[BFSS90] S. Bainbridge, P. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[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 Programming: 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.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Springer-Verlag, 1997.

[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 Programming: 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 βη-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.

[CM98] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[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.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[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.

[DLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. An earlier version appeared in Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL), 1977, pp. 206-214.

[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.

[FF98] Matthias Felleisen and Daniel P. Friedman. *A little Java, A few Patterns*. The MIT Press, Cambridge, Massachusetts, 1998.

[Fre89] Peter J. Freyd. POLYNAT in PER. In John W. Gray and Andre Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 67–68, Providence, Rhode Island, 1989. American Mathematical Society.

[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.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[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.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, MA, 1992.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [Har96].

[Har96] Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See [Har94].

[Hay91] S. Hayashi. Singleton, union and intersection types for program extraction. In T.Ito and A.R.Meyer, editors, *Theoretical Aspect of Computer Software (Lecture Notes in Comuter Science No. 526)*, pages 701–730, Berlin, 1991. Springer-Verlag.

[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.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.

[HJW$^+$92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language haskell, version 1.2. *Sigplan*, 27(5), May 1992.

[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.

[HO94] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. Submitted for publication; electronic draft available through `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Luke.Ong/`, 1994.

[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.

[HP95] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 186–197. ACM, January 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[HP98] Martin Hofmann and Benjamin C. Pierce. Type destructors. In Didier Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998. Full version to appear in *Theory and Practice of Object Systems*.

[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.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999.

[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.

[JMP94] L.S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.

[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.

[KS92] Dinesh Katiyar and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[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.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 142–153, San Francisco, California, January 1995.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, September 1996.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. In John Hughes, editor, *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, 1991.

[LP99] Michael Y. Levin and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. Technical Report MS-CIS-99-19, Dept of CIS, University of Pennsylvania, July 1999.

[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.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Submitted to the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[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.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its metatheory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

[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.

[OR94] Peter W. O'Hearn and Jon G. Riecke. Fully abstract translations and parametric polymorphism. In *ESOP '94: 6th European Symposium on Programming*, April 1994. Springer Lecture Notes in Computer Science, volume 788.

[PA93] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 361–375, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Pie90] Benjamin C. Pierce. Preliminary investigation of a calculus with intersection and union types. Unpublished manuscript, June 1990.

[Pie91a] 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.

[Pie91b] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[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.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pol96] Erik Poll. Width-subtyping and polymorphic record update. Manuscript, June 1996.

[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.

[Pot97] François Pottier. Simplifying subtyping constraints. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1997.

[Pot98] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 228–238, September 1998.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[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).

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990.

[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).

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[Rey84] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Rey98] John Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RP] John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic lambda calculus. Submitted to Information and Computation. Also available as CMU School of Computer Science technical report number CMU-CS-90-147.

[RR94] E. P. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 364–371, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Conference Record of POPL '97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, January 15–17, 1997. ACM Press. Full version to appear in *Theory and Practice of Object Systems, 1998*.

[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.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[TB98] Tofte and Birkedal. A region inference algorithm. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. Addison Wesley, 1991.

[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.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction (vol I)*. North-Holland, 1988.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, San Diego, California, 1995.

[vBM97] J. F. A. K. van Benthem and Alice Ter Meulen, editors. *Handbook of Logic and Language*. MIT Press, 1997.

[Wad90] Philip Wadler. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, 1991.

[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.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[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