Type systems for programming languages

Didier Rémy

Academic year 2013-1024
Version of November 24, 2013
## Contents

1 Introduction 7

1.1 Overview of the course ........................................... 7
1.2 Requirements .................................................. 9
1.3 About Functional Programming ................................. 9
1.4 About Types .................................................. 9
1.5 Acknowledgment ................................................ 11

2 The untyped $\lambda$-calculus 13

2.1 Syntax ..................................................... 13
2.2 Semantics .................................................. 15
   2.2.1 Strong v.s. weak reduction strategies ................... 15
   2.2.2 Call-by-value semantics ................................ 16
2.3 Answers to exercises ......................................... 18

3 Simply-typed lambda-calculus 21

3.1 Syntax ..................................................... 21
3.2 Dynamic semantics ........................................... 21
3.3 Type system ................................................ 22
3.4 Type soundness ................................................
   3.4.1 Proof of subject reduction .............................. 26
   3.4.2 Proof of progress ...................................... 28
3.5 Normalization ................................................ 30
3.6 Simple extensions ............................................
   3.6.1 Unit .................................................. 32
   3.6.2 Boolean ............................................... 32
   3.6.3 Pairs .................................................. 33
   3.6.4 Sums .................................................. 33
   3.6.5 Modularity of extensions ............................... 34
   3.6.6 Recursive functions ................................... 35
   3.6.7 A derived construct: let-bindings ..................... 35
3.7 Exceptions .................................................. 36
## CONTENTS

3.7.1 Semantics ......................................................... 37
3.7.2 Typing rules ....................................................... 38
3.7.3 Variations ......................................................... 38
3.8 References .......................................................... 40
  3.8.1 Language definition ............................................. 41
  3.8.2 Type soundness .................................................. 42
  3.8.3 Tracing effects with a monad .................................. 44
  3.8.4 Memory deallocation .......................................... 44
3.9 Ommitted proofs and answers to exercises .......................... 45

4 Polymorphism and System F ........................................ 49
  4.1 Polymorphism ....................................................... 49
  4.2 Polymorphic $\lambda$-calculus ................................... 51
    4.2.1 Types and typing rules ..................................... 51
    4.2.2 Semantics ..................................................... 52
    4.2.3 Extended System F with datatypes .......................... 54
  4.3 Type soundness ................................................... 58
  4.4 Type erasing semantics ......................................... 62
    4.4.1 Implicitly-typed System F .................................... 62
    4.4.2 Type instance ................................................ 64
    4.4.3 Type containment in System F ............................... 66
    4.4.4 A definition of principal typings ........................... 68
    4.4.5 Type soundness for implicitly-typed System F ............... 68
  4.5 References ........................................................ 72
    4.5.1 A counter example ............................................ 73
    4.5.2 Internalizing configurations ................................. 74
  4.6 Damas and Milner’s type system ................................ 77
    4.6.1 Definition ..................................................... 77
    4.6.2 Syntax-directed presentation ................................ 79
    4.6.3 Type soundness for ML ....................................... 82
  4.7 Ommitted proofs and answers to exercises ........................ 84

5 Type reconstruction ................................................ 91
  5.1 Introduction ....................................................... 91
  5.2 Type inference for simply-typed $\lambda$-calculus ................ 92
    5.2.1 Constraints ................................................... 93
    5.2.2 A detailed example ......................................... 94
    5.2.3 Soundness and completeness of type inference ............... 96
    5.2.4 Constraint solving ......................................... 96
  5.3 Type inference for ML ............................................. 98
    5.3.1 Milner’s Algorithm $\mathcal{J}$ ................................ 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>Constraints</td>
<td>99</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Constraint solving by example</td>
<td>103</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Type reconstruction</td>
<td>106</td>
</tr>
<tr>
<td>5.4</td>
<td>Type annotations</td>
<td></td>
</tr>
<tr>
<td>5.4.1</td>
<td>Explicit binding of type variables</td>
<td>109</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Polymorphic recursion</td>
<td>113</td>
</tr>
<tr>
<td>5.4.3</td>
<td>mixed-prefix</td>
<td>114</td>
</tr>
<tr>
<td>5.5</td>
<td>Equi- and iso-recursive types</td>
<td></td>
</tr>
<tr>
<td>5.5.1</td>
<td>Equi-recursive types</td>
<td>115</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Iso-recursive types</td>
<td>117</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Algebraic data types</td>
<td>118</td>
</tr>
<tr>
<td>5.6</td>
<td>(HM(X))</td>
<td>119</td>
</tr>
<tr>
<td>5.7</td>
<td>Type reconstruction in System F</td>
<td></td>
</tr>
<tr>
<td>5.7.1</td>
<td>Type inference based on Second-order unification</td>
<td>121</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Bidirectional type inference</td>
<td>122</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Partial type inference in MLF</td>
<td>124</td>
</tr>
<tr>
<td>5.8</td>
<td>Proofs and Solution to Exercises</td>
<td>124</td>
</tr>
<tr>
<td>6.1</td>
<td>Towards typed closure conversion</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Existential types</td>
<td></td>
</tr>
<tr>
<td>6.2.1</td>
<td>Existential types in Church style (explicitly typed)</td>
<td>130</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Implicitly-type existential types</td>
<td>133</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Existential types in ML</td>
<td>135</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Existential types in OCaml</td>
<td>136</td>
</tr>
<tr>
<td>6.3</td>
<td>Typed closure conversion</td>
<td></td>
</tr>
<tr>
<td>6.3.1</td>
<td>Environment-passing closure conversion</td>
<td>137</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Closure-passing closure conversion</td>
<td>139</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Mutually recursive functions</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>An overview</td>
<td></td>
</tr>
<tr>
<td>7.1.1</td>
<td>Why use overloading?</td>
<td>145</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Different forms of overloading</td>
<td>146</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Static overloading</td>
<td>147</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Dynamic resolution with a type passing semantics</td>
<td>147</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Dynamic overloading with a type erasing semantics</td>
<td>148</td>
</tr>
<tr>
<td>7.2</td>
<td>Mini Haskell</td>
<td></td>
</tr>
<tr>
<td>7.2.1</td>
<td>Examples in (MH)</td>
<td>149</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The definition of Mini Haskell</td>
<td>150</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Semantics of Mini Haskell</td>
<td>152</td>
</tr>
</tbody>
</table>
7.2.4 Elaboration of expressions ........................................... 154
7.2.5 Summary of the elaboration ........................................... 155
7.2.6 Elaboration of dictionaries ........................................... 157
7.3 Implicitly-typed terms .................................................. 159
7.4 Variations ............................................................... 165
7.5 Ommitted proofs and answers to exercises ......................... 169
Chapter 1

Introduction

These are course notes for part of the master course *Typing and Semantics of functional Programming Languages* taught at the MPRI (Parisian Master of Research in Computer Science) in 2010, 2011, 2012.

The aim of the course is to provide students with the basic knowledge for understanding modern programming languages and designing extensions of existing languages or new languages. The course focuses on the semantics of programming languages.

We present programming languages formally, with their syntax, type system, and operational semantics. We then prove soundness of the semantics, i.e. that well-typed programs cannot go wrong. We do not study full-fledged languages but their core calculi, from which other constructions can be easily added. The underlying computational language is the untyped λ-calculus, extended with primitives, store, etc.

1.1 Overview of the course

These notes only cover part of the course, described below in the paragraph *Typed languages*. Here, we give a brief overview of the whole course to put the study of *Typed languages* into perspective.

**Untyped languages.** Although all the programming languages we study are typed, their underlying computational model is the untyped λ-calculus. That is, types can be dropped after type checking and before evaluation.

Therefore, the course starts with a few reminders about the untyped λ-calculus, even though those are assumed to be known. We show how to extend the pure λ-calculus with constants and primitives and a few other constructs to make it a small programming language. This is also an opportunity to present source program transformations and compi-
lution techniques for function languages, which do not depend much on types. This part is taught by Xavier Leroy.

**Typed languages** Types play a central role in the design of modern programming languages, so they also play a key role in this course. In fact, once we restrict our study to functional languages, the main differences between languages lie more often in the differences between their type systems than between other aspects of their design.

Hence, the course is primarily structured around type systems. We remind the simply-typed \( \lambda \)-calculus, the simplest of type systems for functional languages, and show how to extend it with other fundamental constructs of programming languages.

We introduce polymorphism with System F. We present ML as a restriction of System F for which type reconstruction is simple and efficient. We actually introduce a slight generalization HM(X) of ML to ease and generalize the study of type reconstruction for ML. We discuss techniques for type reconstruction in System F—but without formalizing the details.

We present existential types, first in the context of System F, and then discuss their integration in ML.

Finally, we study the problem of overloading. Overloading differs from other language constructs as the semantics of source programs depend on their types, even though types should be erased at runtime! We thus use overloading as an example of elaboration of source terms, whose semantics is typed, into an internal language, whose semantics is untyped.

**Towards program proofs** Types, as in ML or System F, ensure type soundness, *i.e.* that programs do not go wrong. However useful, this remains a weak property of programs. One often wishes to write more accurate specifications of the actual behavior of programs and prove the implementation correct with respect to them. Finer invariants of data-structures may be expressed within types using Generalized Algebraic Data Types (GADT); or one step further using dependent types. However, one may also describe the behavior of programs outside of proper types *per se*, by writing logic formulas as pre and post conditions, and verifying them mechanically, *e.g.* with a proof assistant. This spectrum of solutions will be presented by Yann Regis-Gianas.

**Subtyping and recursive types** The last part of the course, taught by Giuseppe Castagna, focuses on subtyping, and in particular on semantic subtyping. This allows for very precise types that can be used to describe semi-structured data. Recursive types are also presented in this context, where they play a crucial role.
1.2 Requirements

We assume the reader familiar with the notion of programming languages. Some experience of programming in a typed functional language such as ML or Haskell will be quite helpful. Some knowledge in operational semantics, $\lambda$-calculus, terms, and substitutions is needed. The reader with missing background may find relevant chapters in the book *Types And Programming Languages* by Pierce (2002).

1.3 About Functional Programming

The term *functional programming* means various things. Functional programming views functions as ordinary data which, in particular, can be passed as arguments to other functions and stored in data structures.

A common idea behind functional programming is that repetitive patterns can be abstracted away as functions that may be called several times so as to avoid code duplication. For this reason, functional programming also often loosely or strongly discourages the use of modifiable data, in favor of effect-free transformations of data. (In contrast, the mainstream object-oriented programming languages view objects as the primary kind of data and encourage the use of modifiable data.)

Functional programming languages are traditionally typed (Scheme and Erlang are exceptions) and have close connections with logic. We will focus on typed languages. Because functional programming puts emphasis on reusability and sharing multiple uses of the same code, even in different contexts, they require and make heavy use of polymorphism; when programming in the large, abstraction over implementation details relies on an expressive module system. Types unquestionably play a central role, as explained next.

Functional programming languages are usually given a precise and formal semantics derived from the one of the $\lambda$-calculus. The semantics of languages differ in that some are strict (ML) and some are lazy (Haskell) Hughes (1989). This difference has a large impact on the language design and on the programming style, but has usually little impact on typing.

Functional programming languages are usually sequential languages, whose model of evaluation is not concurrent, even if core languages may then be extended with primitives to support concurrency.

1.4 About Types

A *type* is a concise, formal description of the behavior of a program fragment. For instance, int describes an expression that evaluates to an integer; int → bool describes a function that maps an integer argument to a boolean result; (int → bool) → (list int → list int) describes a function that maps an integer predicate to an integer list transformer.
Types must be sound. That is, programs must behave as prescribed by their types. Hence, types must be checked and ill-typed programs must be rejected.

Types are useful for quite different reasons: They first serve as machine-checked documentation. More importantly, they provide a safety guarantee. As stated by Milner (1978), “Well-typed expressions do not go wrong.” Advanced type systems can also guarantee various forms of security, resource usage, complexity, etc. Types encourage separate compilation, modularity, and abstraction. Reynolds (1983) said: “Type structure is a syntactic discipline for enforcing levels of abstraction.” Types can be abstract. Even seemingly non-abstract types offer a degree of abstraction. For example, a function type does not tell how a function is represented at the machine level. Types can also be used to drive compiler optimizations.

Type-checking is compositional: type-checking an application depends on the type of the function and the type of the argument and not on their code. This is a key to modularity and code maintenance: replacing a function by another one of the same type will preserve well-typedness of the whole program.

Type-preserving compilation Types make sense in low-level programming languages as well—even assembly languages can be statically typed! as first popularized by Morrisett et al. (1999). In a type-preserving compiler, every intermediate language is typed, and every compilation phase maps typed programs to typed programs. Preserving types provides insight into a transformation, helps debug it, and paves the way to a semantics preservation proof (Chlipala, 2007). Interestingly enough, lower-level programming languages often require richer type systems than their high-level counterparts.

Typed or untyped? Reynolds (1985) nicely sums up a long and rather acrimonious debate: “One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility.” A sound type system with decidable type-checking (and possibly decidable type inference) must be conservative.

Later, Reynolds also settles the debate: “From the theorist’s point of view, both sides are right, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages.”

Today, the question is rather whether to use basic types (e.g. as in ML or System F) or sophisticated types (e.g. with dependent types, logical assertions, affine types, capabilities and ownership, etc.) or full program proofs as in the compcert project (Leroy, 2006)!

Explicit v.s. implicit types? The typed v.s. untyped flavor of a programming language should not be confused with the question of whether types of a programming language are explicit or implicit.
Annotating programs with types can lead to a lot of redundancies. Types can even become extremely cumbersome when they have to be explicitly and repeatedly provided. In some pathological cases, they may even increase the size of source terms non linearly. This creates a need for a certain degree of type reconstruction (also called type inference), where the source program may contain some—but not all—type information.

When the semantics is untyped, i.e. types could in principle be entirely left implicit, even if the language is typed. A well-typed program is then one that is the type erasure of a (well-typed) explicitly-typed program. However, full type reconstruction is undecidable for expressive type systems, leading to partial type reconstruction algorithms.

An important issue with type reconstruction is its robustness to small program changes. Because type systems are compositional, a type inference problem can often be expressed as a constraint solving problem, where constraints are made up of predicates about types, conjunction, and existential quantification.

1.5 Acknowledgment

These course notes are based on and still contain a lot of material from a previous course taught for several years by François Pottier.
CHAPTER 1. INTRODUCTION
Chapter 2
The untyped $\lambda$-calculus

In this course, $\lambda$-calculus is the underlying computational language. The $\lambda$-calculus supports natural encodings of many programming languages [Landin (1965)], and as such provides a suitable setting for studying type systems. Following Church’s thesis, any Turing-complete language can be used to encode any programming language. However, these encodings might not be natural or simple enough to help us in understanding their typing discipline. Using $\lambda$-calculus, most of our results can also be applied to other languages (Java, assembly language, etc.).

The untyped $\lambda$-calculus and its extension with the main constructs of programming languages have been presented in the first part of the course taught by Xavier Leroy. Hereafter, we just recall some of the notations and concepts used in our part of the course.

2.1 Syntax

We assume given a denumerable set of term variables, denoted by letter $x$. Then $\lambda$-terms, also known as terms and expressions, are given by the grammar:

$$a ::= x \mid \lambda x. a \mid a \ a \ldots$$

This definition says that an expression $a$ is a variable $x$, an abstraction $\lambda x. a$, or an application $a_1 \ a_2$. The “...” is just a place holder for more term constructs that will be introduced later on. Formally, the “...” is taken empty in the current definition of expressions. However, we may later extend expressions, for instance with let-bindings using the meta-notation:

$$a ::= \ldots \mid \text{let } x = a \text{ in } a$$

which means that the new set of expressions is to be understood as:

$$a ::= x \mid \lambda x. a \mid a \ a \mid \text{let } x = a \text{ in } a$$

The expression $\lambda x. a$ binds variable $x$ in $a$. We write $[x \mapsto a_0]a$ for the capture avoiding substitution of $a_0$ for $x$ in $a$. Terms are considered equal up to the renaming of bound
variables. That is $\lambda x_1.\lambda x_2. x_1 \ (x_1 \ x_2)$ and $\lambda y. \lambda x. y \ (y \ x)$ are really the same term. And $\lambda x. \lambda x. a$ is equal to $\lambda y. \lambda x. a$ when $y$ does not appear free in $a$.

When inspecting the structure of terms, we often need to open up a $\lambda$-abstraction $\lambda x. a$ to expose its body $a$. Then, $a$ usually contains free occurrences of $x$ (that were bound in $\lambda x. a$). When doing so, we may assume, w.l.o.g.\[1\] that $x$ is fresh for (i.e. does not appear free in) any given set of finite variables.

Concrete v.s. abstract syntax  For our meta-theoretical study, we are interested in the abstract syntax of expressions rather than their concrete syntax. Hence, we like to think of expressions as their abstract syntax trees. Still, we need to write expressions on paper, i.e. strings of characters, hence we need some concrete syntax for terms. The compromise is to have some concrete syntax that is in one-to-one correspondence with the abstract syntax.

An expression in concrete notation, e.g. $\lambda x. \lambda y. x \ y$ must be understood as its abstract syntax tree (next on the right).

For convenience, we may sometimes introduce syntactic sugar as shorthand; it should then be understood by its expansion into some primitive form. For instance, we may introduce multi-argument functions $\lambda xy. a$ as a short hand for $\lambda x. \lambda y. a$ just for conciseness of notation on paper or readability of examples, but without introducing a new form of expressions into the abstract syntax. (Although, studying multi-parameter functions would also be possible, and then this would not be syntactic sugar, but this is not the route we take here.)

When studying programming languages formally, the core language is usually kept as small as possible avoiding the introduction of new constructs that can already be expressed with existing ones—or are trivial variations on existing ones. Indeed, redundant constructs often obfuscate the essence of the semantics of the language.

Exercise 1 Write a datatype term to represent the abstract syntax of the untyped $\lambda$-calculus.
(Solution p. 18)

Exercise 2 Higher Order Abstract Syntax (HOAS) uses the binding and $\alpha$-conversion mechanisms of the host language (here OCaml) to implement bindings and $\alpha$-conversion of the concrete language. The parametric version of HOAS is moreover parameterized by the type of variables.

```ocaml
type 'a pterm =
| PVar of 'a
| PFun of ('a -> 'a pterm)
| PApp of 'a pterm * 'a pterm
```

\[1\]without lost of generality.
2.2. SEMANTICS

For example, we may define

\[
\text{let } h = \text{PApp} (\text{PFun} (\text{fun } f \rightarrow \text{PApp} (\text{PVar} f, \text{PVar} f)), \text{PFun} (\text{fun } x \rightarrow \text{PVar} x))
\]

Notice that \( h \) is polymorphic in the type of term variables. What term of the \( \lambda \)-calculus does it represent? (Solution p. 15)

Write a function `to_term` that translates from terms in PHOAS (of type `pterm`) into terms in concrete syntax (of type `term`). (Solution p. 18)

2.2 Semantics

The semantics of the \( \lambda \)-calculus is given by a small-step operational semantics, i.e. a reduction relation between \( \lambda \)-terms. It is also called the dynamic semantics since it describes the behavior of programs at runtime, i.e. when programs are executed.

2.2.1 Strong v.s. weak reduction strategies

For the pure \( \lambda \)-calculus, one can choose a strong notion of reduction, i.e. that can perform reduction in any context, in particular under \( \lambda \)-abstractions. This implies that a term can be reduced in many different ways, depending on which redex is reduced first. Despite this, reduction in the \( \lambda \)-calculus is confluent: for terms that are strongly normalizing, i.e. do not contain infinite reduction path, then all possible reduction paths end up on the same normal form: the calculus is confluent.

By contrast, programming languages are usually give a weak reduction strategy, i.e. reduction does not occur under abstractions. The main reason for this choice is simplicity and efficiency or reduction.

The most commonly used strategy is call-by-value, where arguments are reduced before being substituted for the formal parameter of functions. However, some languages also use a call-by-name strategy that delays the evaluation of arguments until they are actually used. In fact, rather than call-by-name, one use implements a call-by-need strategy, which as call-by-name delays the evaluation of arguments, but as call-by-value shares this evaluation: that is, the occurrence of an argument that is used requires its evaluation, but all other occurrences of the argument see the result of the evaluation and do not have to reevaluate the argument if needed. This is however more delicate to formalize and one often uses call-by-name semantics as an approximation of call-by-need semantics.

Although programming languages implement weak reduction strategies, it would make perfect sense to define their semantics in two steps, first using strong notion of reduction, and then restricting the reduction paths to model the actual strategy. The first semantics can then be used to model some program transformations, such as partial evaluation, done at compile time. Another advantage of this approach is that weak reduction strategies are a particular case of strong reduction strategies. Hence, (positive) properties
can be established once for all for strong reduction and will also hold for weak reduction strategies, including both call-by-value and call-by-name.

However, as soon as we extend the language with impure constructs such as side effects or exceptions, it does not make much sense to define those in combination with strong reduction strategies. Even in a pure setting and for instance a final call-by-value operation semantics doing partial evaluation (i.e. strong reduction) at compile time may alter the semantics of programs because of possible non-termination. Hence, such transformation can only be perform for part of programs that are total (always terminal without an error).

Finally, the formalization is often simpler when working with weak reductions strategies. Thus, despite some advantages of the two step-approach to the semantics of programming languages, we will not follow this approach here. Instead we directly start with a weak reduction strategy. Still, we will informally discuss at certain places some of the properties that would hold if we had followed such the more general approach.

2.2.2 Call-by-value semantics

We choose a call-by-value semantics. When explaining references, exceptions, or other forms of side effects, this choice matters. Otherwise, most of the type-theoretic machinery applies to call-by-name or call-by-need—actually to any weak reduction strategy—just as well.

In the pure \(\lambda\)-calculus, the values are the functions:

\[ v ::= \lambda x. a | \ldots \]

Variables are not values in the call-by-value \(\lambda\)-calculus. We only evaluate closed terms, hence a variable should never appear in an evaluation context. Notice that any function is a value in the call-by-value \(\lambda\)-calculus, in particular, \(a\) is an arbitrary term. In a strong reduction setting, we could also evaluate the body of the function \(a\), and then, \(a\) should thus not contain any \(\beta\)-redex.

The reduction relation \(a_1 \rightarrow a_2\) is inductively defined:

\[
\begin{align*}
\beta_v & (\lambda x. a) v \rightarrow [x \mapsto v]a \\
\text{Context} & a \rightarrow a' \\
\text{Context} & e[a] \rightarrow e[a']
\end{align*}
\]

\([x \mapsto V]\) is the capture avoiding substitution of \(V\) for \(x\). We write \([x \mapsto V]a\) its application to a term \(a\). Evaluation may only occur in call-by-value evaluation contexts, defined as follows:

\[ e ::= [] | a | v | \ldots \]

Notice that we only need evaluation contexts of depth one, thanks to repeated applications of Rule Context. An evaluation context of arbitrary depth may be defined as a stack of one-hole contexts:

\[ \bar{e} ::= [] | e[\bar{e}] \]
2.2. SEMANTICS

Exercise 3 Define the semantics of the call-by-name $\lambda$-calculus. 
(Solution p. 18)

Exercise 4 Give a big-step operational semantics for the call-by-value $\lambda$-calculus. Compare it with the small-step semantics. What can you say about non terminating programs? How can this be improved? 
(Solution p. 19)

Exercise 5 Write an interpreter for a call-by-value $\lambda$-calculus. Modify the interpreter to have a call-by-name semantics; then a call-by-need semantics. You may instrument the evaluation to count the number of evaluation steps.

Recursion

Recursion is inherent in $\lambda$-calculus, hence reduction may not terminate. For example, the term $(\lambda x. x x) (\lambda x. x x)$ known as $\Delta$ reduces to itself, and so may reduce forever.

A slight variation on $\Delta$ is the fix-point combinator $Y$, defined as

$$\lambda g. (\lambda x. x x) (\lambda z. g (z z))$$

Whenever applied to a functional $G$, it reduces in a few steps to $G (Y G)$, which is not yet a value. In a call-by-value setting, this term actually reduces forever—before even performing any interesting computation step. Therefore, we instead use its $\eta$-expanded version $Z$ that guards the duplication of the generator $G$:

$$\lambda g. (\lambda x. x x) (\lambda z. g (\lambda v. z z v))$$

Exercise 6 Check that $Y G$ reduces for ever. Check that $Z G$ does not. Check that $Z G v$ behaves as expected—unfolds the recursion after the body of $G$ has been evaluated.

Exercise 7 Define the fixpoint combination $Z$ in OCaml—without using let rec. Why do you need the $-rectype$ option? Use $Z$ to define the factorial function (still without using let rec).

(Solution p. 19)
2.3 Answers to exercises

Solution of Exercise 1

\[
\text{type } \text{var} = \text{string} \\
\text{type } \text{term} = \\
\quad \mid \text{Var of } \text{var} \\
\quad \mid \text{Fun of } \text{var} \times \text{term} \\
\quad \mid \text{App of } \text{term} \times \text{term}
\]

Define in this abstract syntax the term \text{funaa}.

Solution of Exercise 2

\[(\lambda f. f f)(\lambda x. x).\]

Solution of Exercise 2, Question 2

\[
\begin{align*}
\text{let gensym = let } n = \text{ref } 0 \text{ in fun } () \rightarrow \text{incr } n; \text{"x"} \times \text{string of int } \text{!n};; \\
\text{let rec } \text{to_term} = \text{function} \\
\quad \mid \text{PFun } f \rightarrow \text{let } x = \text{gensym()} \text{ in Fun } (x, \text{to_term } (f (\text{Var } x))) \\
\quad \mid \text{PApp } (f, g) \rightarrow \text{App } (\text{to_term } f, \text{to_term } g) \\
\quad \mid \text{PVar } x \rightarrow x \\
\text{let } t = \text{to_term } h
\end{align*}
\]

\[
\text{val } t : \text{term} = \text{App } (\text{Fun } \text{"x2"}, \text{App } (\text{Var } \text{"x2"}, \text{Var } \text{"x2"})), \text{Fun } \text{"x1"}, \text{Var } \text{"x1"})
\]

Solution of Exercise 3

Values are unchanged. Evaluation contexts only allow the evaluation in function position:

\[e ::= [] a\]

As a counterpart, \(\beta\)-reduction must not require its argument to be evaluated. Hence the call-by-name \(\beta_n\) rule is:

\[(\lambda x. a_0) a \rightarrow [x \mapsto a] a_0 \quad (\beta_n)\]
2.3. ANSWERS TO EXERCISES

Solution of Exercise 4

The big step semantics defines an evaluation relation $\mathcal{E} \vdash a \sim v$ where $\mathcal{E}$ is an evaluation environment $\mathcal{E}$ that maps variables to values. The relation is defined by inference rules:

\[
\begin{align*}
\text{Eval-Fun} & : \quad \mathcal{E} \vdash \lambda x. a \sim \lambda x. a \\
\text{Eval-Var} & : \quad x \mapsto v \in \mathcal{E} \quad \mathcal{E} \vdash x \sim v \\
\text{Eval-App} & : \quad \mathcal{E} \vdash a_1 \sim \lambda x. a \quad \mathcal{E} \vdash a_2 \sim v \\
& \quad \mathcal{E}, x \mapsto v \vdash a \sim v \\
& \quad \mathcal{E} \vdash a_1 a_2 \sim v
\end{align*}
\]

Rule $\text{Eval-Fun}$ says that a function is a value and evaluates to itself. Rule $\text{Eval-App}$ evaluates both sides of an application. Provided the left-hand side evaluates to a function $\lambda x. a$, we may evaluation $a$ in an extended context where $x$ is mapped to the evaluation of the right-hand side. The results of the evaluation of $a$ is then the result of the evaluation of the application.

Notice that the definition is partial: if the left-hand side does not evaluate to a function (e.g. it could be a free variable), then the evaluation of the application is not defined. Similarly, the evaluation of a variable that is not bound in the environment is undefined.

Furthermore, the evaluation is also undefined for programs that loops, such as $(\lambda x. x x) (\lambda x. x x)$: one will attempt to build an infinite evaluation derivation, but as this never ends, we cannot formally say anything about its evaluation.

Solution of Exercise 7

The definition contains an auto-application of a $\lambda$-bound variable $\text{fun } x \to x x$. In OCaml, this is ill-typed, as it requires $x$ to have both types $\alpha$ and $\alpha \to \beta$ simultaneously, which is only possible if $\alpha$ is a recursive type $(\ldots(\alpha \to \ldots) \to \alpha)$ With the \texttt{-rectype} option, one can defined:

\[
\begin{align*}
\text{let } zfix g &= (\text{fun } x \to x x) (\text{fun } z \to g (\text{fun } v \to z z v)) \\
\text{let } gfact f n &= \text{if } n > 0 \text{ then } n * f (n-1) \text{ else } 1 \\
\text{let } fact &= zfix gfact; \\
\text{let } six &= fact 3;
\end{align*}
\]

which correctly evaluates $six$ to the integer 6.
Chapter 3

Simply-typed lambda-calculus

This chapter is an introduction to typed languages. The formalization will be subsumed by that of System $F$ in the next chapter. We still give all the definitions and the proofs of the main results in this simpler setting for pedagogical purposes. Their generalization in the more general setting of System $F$ will then be easier to understand.

3.1 Syntax

We give an explicitly typed version of the simply-typed $\lambda$-calculus. Therefore, we modify the syntax of the $\lambda$-calculus to add type annotations for parameters of functions. In order to avoid confusion, we write $M$ instead of $a$ for explicitly typed expressions.

$$M ::= x \mid \lambda x : \tau. M \mid M M \mid \ldots$$

As earlier, the “…” are a place holder for further extensions of the language. Types are denoted by letter $\tau$ and defined by the following grammar:

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \ldots$$

where $\alpha$ denotes a type variable. We assume given a denumerable collection of type variables. This definition says that a type $\tau$ is a type variable $\alpha$, or an arrow type $\tau_1 \rightarrow \tau_2$.

3.2 Dynamic semantics

The dynamic semantics of the simply-typed $\lambda$-calculus is obtained by modifying the dynamic semantics of the $\lambda$-calculus in the obvious way to accommodate for type annotations of function parameters, which are just ignored. Values and evaluation contexts become:

$$V ::= \lambda x : \tau. M \mid \ldots \quad E ::= [] M \mid V [] \mid \ldots$$
The reduction relation $M_1 \rightarrow M_2$ is inductively defined by:

$$
\beta_v 
(\lambda x: \tau. M) \ V \rightarrow [x \mapsto V] M
$$

$$
\text{Context} 
M \rightarrow M' 
E[M] \rightarrow E[M']
$$

The semantics of simply-typed $\lambda$-calculus is obviously type erasing, i.e., as we shall see in the next section (§3.3).

### 3.3 Type system

In typed $\lambda$-calculi, not all syntactically well-formed programs are accepted—only well-typed programs are. Well-typedness is defined as a 3-place predicate $\Gamma \vdash M : \tau$ called a typing judgment.

The typing context $\Gamma$ (also called a typing environment) is a finite sequence of bindings of program variables to types. The empty context is written $\emptyset$. A typing context $\Gamma$ can be extended with a new binding $\tau$ for $x$ with the notation $\Gamma, x : \tau$. To avoid confusion between the new binding and any other binding that may appear in $\Gamma$, we disallow extending contexts to bind the same variable several times. This is not restrictive because bound variables can always be renamed in source programs to avoid name clashes. A typing context can then be thought of as a finite function from program variables to their types. We write $\text{dom}(\Gamma)$ for the set of variables bound by $\Gamma$ and $\Gamma(x)$ for the type $\tau$ bound to $x$ in $\Gamma$, which implies that $x$ is in $\text{dom}(\Gamma)$. We write $x : \tau \in \Gamma$ to mean that $\Gamma$ maps $x$ to $\tau$, and $x \not\in \text{dom}(\Gamma)$ to mean that $x \not\in \text{dom}(\Gamma)$.

Typing judgments are defined inductively by the following inference rules:

$$
\text{Var} 
\frac{\Gamma \vdash x : \Gamma(x)}{
\Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2}
$$

$$
\text{Abs} 
\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{
\Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2}
$$

$$
\text{App} 
\frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 
\Gamma \vdash M_2 : \tau_1}{
\Gamma \vdash M_1 \ M_2 : \tau_2}
$$

By our convention on well-formedness of typing contexts, the premise of rule $\text{Abs}$ carries the implicit assumption $x \not\in \text{dom}(\Gamma)$. This condition can always be satisfied, since $x$ is bound in the expression $\lambda x : \tau. M$ and can be renamed if necessary.

Notice that the specification is extremely simple. In the simply-typed $\lambda$-calculus, the definition is syntax-directed. That is, at most one rule applies for an expression; hence, the shape of the derivation tree for proving a judgment $\Gamma \vdash M : \tau$ is fully determined by the shape of the expression $M$. This is not true of all type systems.

A typing derivation is a proof tree that witnesses the validity of a typing judgment: each node is the application of a typing rule. A proof tree is either a single node composed of an axiom (a typing rule without premises) or a typing rule with as many proof-subtrees as typing judgment premises.

For example, the following is a typing derivation for the compose function in the empty
environment where $\Gamma$ stands for $f : \tau_1 \rightarrow \tau_2; g : \tau_0 \rightarrow \tau_1; x : \tau_0$.

This derivation is valid for any choice of $\tau_1$ and $\tau_2$. Conversely, every derivation for this term must have this shape, for some $\tau_1$ and $\tau_2$.

This suggests a procedure for type inference: build the shape of the derivation from the shape of the expression. Then, solve the constraints on types so that the derivation is valid. This informal procedure to search for possible derivations is justified formally by the inversion lemma, which describes how the subterms of a well-typed term can be typed.

Lemma 1 (Inversion of typing rules) Assume $\Gamma \vdash M : \tau$.

- If $M$ is a variable $x$, then $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = \tau$.

- If $M$ is $M_1 M_2$ then $\Gamma \vdash M_1 : \tau_2 \rightarrow \tau$ and $\Gamma \vdash M_2 : \tau_2$ for some type $\tau_2$.

- If $M$ is $\lambda x : \tau_0. M_1$, then $\tau$ is of the form $\tau_0 \rightarrow \tau_1$ and $\Gamma, x : \tau_0 \vdash M_1 : \tau_1$.

The inversion lemma is a basic property that is used in many places when reasoning by induction on terms. Although trivial in our simple setting, stating it explicitly avoids informal reasoning in proofs; in more general settings, this may be a difficult lemma that requires reorganizing typing derivations.

In our settings, the typing rules are syntax-directed. That is, for any given well-formed expression, at most one typing rule may apply. Then, the shape of the typing derivation tree is unique and fully determined by the shape of the term.

Moreover, each term has actually a unique type. Hence, typing derivations are unique, in a given typing context. The proof is a straightforward induction on the structure of terms.

Explicitly-typed terms can thus be used to describe typing derivations (up to the typing context) in a precise and concise way, because terms of the language have a concrete syntax. This enables reasoning by induction on terms, which is often lighter than reasoning by induction on typing derivations, since terms are concrete objects while derivations are in the meta-language of mathematics.

This also makes typechecking a trivial recursive function that checks that for each expression that the unique candidate typing rule can be correctly instantiated.

Of course, the existence of syntax-directed typing rules relies on type information present in source terms. Uniqueness of typing derivations can be easily lost if some type information
is left implicit. At some extreme, types may be left implicit and only appear in typing derivations; then there would be many possible derivations for the same term.

Explicitly v.s. implicitly typed? Our presentation of simply-typed $\lambda$-calculus is explicitly typed (we also say in church-style), as parameters of abstractions are annotated with their types. Simply-typed $\lambda$-calculus can also be implicitly typed (we also say in curry-style) when parameters of abstractions are left unannotated, as in the plain $\lambda$-calculus.

We may easily translate explicitly-typed expressions into implicitly-typed ones by dropping type annotations. This is called type erasure. We write $\lceil M \rceil$ for the type erasure of $M$, which is defined by structural induction on $M$:

\[
\begin{align*}
\lceil x \rceil & \triangleq x \\
\lceil \lambda x : \tau . \mathcal{M} \rceil & \triangleq \lambda x . \lceil \mathcal{M} \rceil \\
\lceil \mathcal{M}_1 \mathcal{M}_2 \rceil & \triangleq \lceil \mathcal{M}_1 \rceil \lceil \mathcal{M}_2 \rceil
\end{align*}
\]

The erasure of a term $\mathcal{M}$ of System $\mathsf{F}$ is an untyped $\lambda$-term $a$.

Conversely, can we convert implicitly-typed expressions back into explicitly-typed ones, that is, can we reconstruct the missing type information? This is equivalent to finding a typing derivation for implicitly-typed terms. It is called type reconstruction (or type inference and is much more involved than just type-checking explicitly typed terms—see the chapter on type inference (§5).

Untyped semantics Observe that although the reduction carries types at runtime, types do not actually contribute to the reduction. Intuitively, the semantics of terms is the same as that of their type erasure.

Formally, we must be more careful, as terms and their erasure do not live in the same world. Instead, we may say that the two semantics coincide by putting them into correspondence.

The semantics is said to be untyped or type-erasing if any reduction step on source terms can be reproduced in the untyped language between their type erasures (forward simulation), and conversely, a reduction step after type erasure can also be traced back in the typed language as a reduction step between associated source terms (backward simulation). Formally, this can be stated as follows:

**Lemma 2 (forward simulation)** If $M_1 \rightarrow M_2$ then $\lceil M_1 \rceil \rightarrow \lceil M_2 \rceil$.

**Lemma 3 (backward simulation)** If $\lceil M \rceil \rightarrow a$, then there exists $M'$ such that $M \rightarrow M'$ and $\lceil M' \rceil = a$. 
Diagramatically, we have

\[ M_1 \xrightarrow{\beta} M_2 \]

\[ a_1 \xrightarrow{\beta} a_2 \]

\[ \overrightarrow{\beta} \]

Forward simulation

\[ M_1 \xleftrightarrow{\beta} M_2 \]

\[ a_1 \xleftrightarrow{\beta} a_2 \]

\[ \overleftarrow{\beta} \]

Backward simulation

The combination of both lemmas establishes a *bisimulation* between explicitly-typed terms and implicitly-typed ones.

In our simple setting this is a one-to-one correspondence, and the proof is immediate and not very interesting. The proof will be done in the more general case of System F.

In general, there may be reduction steps on source terms that involve only types and that have no counter-part on compiled terms. This extra flexibility between the two semantics is often useful. It can be formalized by splitting the reduction \( \rightarrow \) into \( \iota \)-steps \( \rightarrow_\iota \), which are dropped after type erasure, and \( \beta \)-steps \( \rightarrow_\beta \), which are kept after type erasure. The \( \iota \)-reduction must be terminating.

**Exercise 8** Rewrite the two previous lemmas to allow \( \iota \)-steps. What could happen if \( \iota \)-reduction were not terminating?

(Solution p. 45)

Having a *type-erasing semantics* is an important property of a language: it simplifies its meta-theoretical study since its semantics does not depend on types. It also means that types can be ignored at runtime.

Be aware that an implicitly typed language does not necessarily have a type-erasing semantics. In Haskell, for instance, types drive the semantics via the choice of type classes even though they are inferred. In fact, Haskell surface programs are elaborated by compiling type classes away into an internal typed language which itself has an erasing semantics.

### 3.4 Type soundness

Type soundness is often known as Milner’s slogan “Well-typed expressions do not go wrong.” What is a formal statement of this? By definition, a closed term \( M \) is well-typed if it admits some type \( \tau \) in the empty environment. By definition, a closed, irreducible term is either a value or stuck. A closed term must converge to a value, diverge, or go wrong by reducing to a stuck term. Milner’s slogan now has a formal meaning:

**Theorem 1 (Type Soundness)** *Well-typed expressions do not go wrong.*
The proof of type soundness is by combination of Subject Reduction (Lemma 2) and Progress (Lemma 3). This syntactic proof method is due to Wright and Felleisen (1994).

**Theorem 2 (Subject reduction)** Reduction preserves types: if $M_1 \rightarrow M_2$, then for any type $\tau$ such that $\emptyset \vdash M_1 : \tau$, we also have $\emptyset \vdash M_2 : \tau$.

**Theorem 3 (Progress)** A well-typed, closed term is either reducible or a value: if $\emptyset \vdash M : \tau$, then there exists $M'$ such that $M \rightarrow M'$ or $M$ is a value.

Progress also says that no stuck term is well-typed. We sometimes use an equivalent formulation of progress: a closed, well-typed irreducible term is a value, i.e. if $\emptyset \vdash M : \tau$ and $M \not\rightarrow$ then $M$ is a value.

### 3.4.1 Proof of subject reduction

Subject reduction is proved by induction over the hypothesis $M_1 \rightarrow M_2$. Thus, there is one case per reduction rule. In the pure simply-typed $\lambda$-calculus, there are just two such rules: $\beta$-reduction and reduction under an evaluation context.

**Type preservation by $\beta$-reduction.**

In the proof of subject reduction for the $\beta$-reduction case, the hypotheses are

$$(\lambda x: \tau. M) V \rightarrow [x \mapsto V] M \quad (1) \quad \emptyset \vdash (\lambda x: \tau. M) V : \tau_0 \quad (2)$$

and the goal is $\emptyset \vdash [x \mapsto V] M : \tau_0 \quad (3)$.

To proceed, we decompose the hypothesis (2): by inversion (Lemma 1), its derivation of (2) must be of the form:

$$\begin{align*}
\text{Abs} & \quad x : \tau \vdash M : \tau_0 \quad (4) \\
\text{App} & \quad \emptyset \vdash (\lambda x: \tau. M) : \tau \rightarrow \tau_0 \quad \emptyset \vdash V : \tau \quad (5) \\
\emptyset & \vdash (\lambda x: \tau. M) V : \tau_0 \quad (2)
\end{align*}$$

We expect the conclusion (3) to follow from (4) and (5). Indeed, we could conclude with the following lemma:

**Lemma 4 (Value substitution)** If $x : \tau \vdash M : \tau_0$ and $\emptyset \vdash V : \tau$, then $\emptyset \vdash [x \mapsto V] M : \tau_0$.

In plain words, replacing a formal parameter with a type-compatible actual argument preserves types. Unsurprisingly, this lemma must be suitably generalized so that it can be proved by structural induction over the typing derivation for $M$:

**Lemma 5 (Value substitution, strengthened)** If $x : \tau, \Gamma \vdash M : \tau_0$ and $\emptyset \vdash V : \tau$, then $\Gamma \vdash [x \mapsto V] M : \tau_0$. 
3.4. TYPE SOUNDNESS

The proof is then straightforward provided we have a weakening lemma (stated below) in the case for variables. (In the case for abstraction, the variable for the parameter can—and must—be chosen different from the variable $x$.) This closes the $\beta$-reduction proof case for type preservation.

Exercise 9 Write all the details of the proof of value substitution.

The weakening we have used in the proof of type preservation for $\beta$-reduction is:

**Lemma 6 (Weakening)** If $\emptyset \vdash V : \tau_1$ then $\Gamma \vdash V : \tau_1$.

We may actually prove a simplified version adding only one binding at a time, as the general case follows as a corollary. However, the lemma must also be strengthened.

**Remark 1** Strengthening will often be needed for properties of interest in this course, which are about explicitly-typed terms, or equivalently, typing derivations, and proved by structural induction, i.e. by induction and case analysis on the structure of the term (or its derivation), because well-typedness of subterms may involve a larger typing context than the one used for the inclosing term. Therefore, properties stated for a term $M$ must hold not under a particular context in which $M$ is typed but under all extensions of such a context.

**Lemma 7 (Weakening, strengthened)** If $\Gamma \vdash M : \tau$ and $y \notin \text{dom}(\Gamma)$, then $\Gamma, y : \tau' \vdash M : \tau$.

**Proof**: The proof is by structural induction on $M$, applying the inversion lemma:

*Case $M$ is $x$:* Then $x$ must be bound to $\tau$ in $\Gamma$. Hence, it is also bound to $\tau$ in $\Gamma, y : \tau'$. We conclude by rule $\text{VAR}$.

*Case $M$ is $\lambda x : \tau_2. M_1$:* W.l.o.g., we may choose $x \notin \text{dom}(\Gamma)$ and $x \neq y$. We have $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$ with $\tau_2 \to \tau_1$ equal to $\tau$. By induction hypothesis, we have $\Gamma, x : \tau_2, y : \tau' \vdash M_1 : \tau_1$. Thanks to a permutation lemma, we have $\Gamma, y : \tau', x : \tau_2 \vdash M_1 : \tau_1$ and we conclude by Rule $\text{ABS}$.

*Case $M$ is $M_1 M_2$:* easy.

Exercise 10 Write the details of the application case for weakening.

(Solution p. 45)

Exercise 11 Try to prove the unstrengthened version and see where you get stuck.

(Solution p. 45)

**Lemma 8 (Permutation lemma)** If $\Gamma \vdash M : \tau$ and $\Gamma'$ is a permutation of $\Gamma$, then $\Gamma' \vdash M : \tau$.

The result is obvious since a permutation of $\Gamma$ does not change its interpretation as a finite function, which is all what is used in the typing rules so far (this will no longer be the case when we extend $\Gamma$ with type variable declarations). Formally, the proof is by induction on $M$. 
Type preservation by reduction under an evaluation context.

The first hypothesis is \( M \rightarrow M' \) (1) where, by induction hypothesis, this reduction preserves types (2). The second hypothesis is \( \emptyset \vdash E[M] : \tau \) (3) where \( E \) is an evaluation context. The goal is \( \emptyset \vdash E[M'] : \tau \) (4).

Observe that typechecking is compositional: only the type of the subexpression in the hole matters, not its exact form, as stated by the compositionality Lemma, below. The context case immediately follows from compositionality, which closes the proof of subject reduction.

**Lemma 9 (Compositionality)** If \( \emptyset \vdash E[M] : \tau \), then, there exists \( \tau' \) such that:

- \( \emptyset \vdash M : \tau' \), and
- for every term \( M' \) such that \( \emptyset \vdash M' : \tau' \), we have \( \emptyset \vdash E[M'] : \tau \).

The proof is by cases over \( E \); each case is straightforward.

**Remark 2** Informally, \( \tau' \) is the type of the hole in the context \( E \), itself of type \( \tau \); we could write the pseudo judgment \( \emptyset \vdash E[\tau'] : \tau \). (This judgment could also be defined by formal typing rules, of course.)

### 3.4.2 Proof of progress

Progress (Theorem 3) says that (closed) well-typed terms are either reducible or values. It is proved by structural induction over the term \( M \). Thus, there is one case per construct in the syntax of terms.

In the pure \( \lambda \)-calculus, there are just three cases: variable; \( \lambda \)-abstraction; and application. The case of variables is void, since a variable is never well-typed in the empty environment. The case of \( \lambda \)-abstractions is immediate, because a \( \lambda \)-abstraction is a value. In the only remaining case of an application, we show that \( M \) is always reducible.

Assume that \( \emptyset \vdash M : \tau_1 \) and \( M \) is an application \( M_1 M_2 \). By inversion of typing rules, there exist types \( \tau_1 \) and \( \tau_2 \) such that \( \emptyset \vdash M_1 : \tau_2 \rightarrow \tau_1 \) and \( \emptyset \vdash M_2 : \tau_2 \). By induction hypothesis, \( M_1 \) is either reducible or a value \( V_1 \). If \( M_1 \) is reducible, so is \( M \) because \( [] M_2 \) is an evaluation context and we are done. Otherwise, by induction hypothesis, \( M_2 \) is either reducible or a value \( V_2 \). If \( M_2 \) is reducible, so is \( M \) because \( V_1 [] \) is an evaluation context and we are done. Otherwise, because \( V_1 \) is a value of type \( \tau_1 \rightarrow \tau_2 \), it must be a \( \lambda \)-abstraction by classification of values (Lemma 10 below), so \( V_1 V_2 \) is a \( \beta \)-redex, hence reducible.

Interestingly, the proof is constructive and corresponds to an algorithm that searches for the active redex in a well-typed term.

In the last case, we have appealed to the following property:

**Lemma 10 (Classification of values)** Assume \( \emptyset \vdash V : \tau \). Then,
3.4. TYPE SOUNDNESS

- if $\tau$ is an arrow type, then $V$ is a $\lambda$-abstraction;
- ...

Proof: By cases over $V$:

- if $V$ is a $\lambda$-abstraction, then $\tau$ must be an arrow type;
- ...

Because different kinds of values receive types with different head constructors, this classification is injective, and can be inverted, which gives exactly the conclusion of the lemma.

In the pure $\lambda$-calculus, classification is trivial, because every value is a $\lambda$-abstraction. Progress holds even in the absence of the well-typedness hypothesis, i.e. in the untyped $\lambda$-calculus, because no term is ever stuck!

As the programming language and its type system are extended with new features, however, type soundness is no longer trivial. Most type soundness proofs are shallow but large. Authors are often tempted to skip the “easy” cases, but these may contain hidden traps!

**Warning!** Sometimes, the combination of two features is unsound, even though each feature, in isolation, is sound. This is problematic, because researchers like studying each feature in isolation, and do not necessarily foresee problems with the combination. This will be illustrated in this course by the interaction between references and polymorphism in ML.

In fact, a few such combinations have been implemented, deployed, and used for some time before they were found to be unsound! For example, this happened for call/cc + polymorphism in SML/NJ [Harper and Lilibrige, 1991]; and for mutable records with existential quantification in Cyclone [Grossman, 2006].

**Soundness versus completeness** Because the $\lambda$-calculus is a Turing-complete programming language, whether a program goes wrong is an undecidable property. (Assuming that it is possible to go wrong, i.e., the calculus is not the pure $\lambda$-calculus, since progress holds in $\lambda$-calculus even for untyped programs, as we have noticed above.) As a consequence, any sound, decidable type system must be incomplete, that is, it must reject some valid programs.

Type systems can be compared against one another via encodings, so it is sometimes possible to prove that one system is more expressive than another. However, whether a type system is “sufficiently expressive in practice” can only be assessed via empirical means. It can take a lot of intuition and experience to determine whether a type system is, or is not, expressive enough in practice.
**Exercise 12** The subject reduction is often stated as “reduction preserve typings”. A typing of a term \( M \) is a pair \((\Gamma, \tau)\) such that \(\Gamma \vdash M : \tau\). Define a relation \(\sqsubseteq\) on typings such that \(M \sqsubseteq M'\) means that all typings of \( M \) are also typings of \( M' \). Restate subject reduction using the relation \(\sqsubseteq\) and proof it.

(Solution p. 45)

### 3.5 Normalization

In general, types also ensure termination of programs—as long as no form of recursion in types or terms has been added. Even if one wishes to add recursion explicitly later on, it is an important property of the design that non-termination is originating from the constructs for recursion only and could not occur without it.

The simply-typed \(\lambda\)-calculus is also lifted at the level of types in richer type systems such as System \(F^\omega\); then, the decidability of type-equality depends on the termination of the reduction at the type level.

Proving termination of reduction in fragments of the \(\lambda\)-calculus is often a difficult task because reduction may create new redexes or duplicate existing ones. However, the proof of termination for the simply-typed \(\lambda\)-calculus is simple enough and interesting to be presented here. Notice that our presentation of simply-typed \(\lambda\)-calculus is equipped with a call-by-value semantics, while proofs of termination are usually done with a strong evaluation strategy where reduction can occur in any context.

We follow the proof schema of Pierce (2002), which is a modern presentation in a call-by-value setting of an older proof by Hindley and Seldin (1986). The proof method, which is now a standard one, is due to Tait (1967). It consists in first building the set \(\mathcal{T}_\tau\) of terminating closed terms of type \(\tau\), and then showing that any term of type \(\tau\) is actually in \(\mathcal{T}_\tau\), by induction on terms. Unfortunately, stated as such, this hypothesis is too weak. The difficulty in such cases is usually to find a strong enough induction hypothesis. The solution in this case is to require that terms in \(\mathcal{T}_{\tau_1 \rightarrow \tau_2}\) not only terminate but also terminate when applied to any term in \(\mathcal{T}_{\tau_1}\).

**Definition 1** Let \(\mathcal{T}_\tau\) be defined inductively on \(\tau\) as follows: let \(\mathcal{T}_\alpha\) be the set of closed terms that terminates; let \(\mathcal{T}_{\tau_2 \rightarrow \tau_1}\) be the set of (closed) terms \(M_1\) of type \(\tau_2 \rightarrow \tau_1\) that terminates and such that \(M_1 M_2\) is in \(\mathcal{T}_{\tau_1}\) for any (closed) term \(M_2\) in \(\mathcal{T}_{\tau_2}\).

The set \(\mathcal{T}_\tau\) can be seen as a predicate, i.e. a unary relation. It is called a (unary) logical relation because it is defined inductively on the structure of types. The following proof is then schematic of the use of logical relations.

We state two obvious lemmas to prepare for the main proof. All terms in \(\mathcal{T}_\tau\) terminate, by definition of \(\mathcal{T}_\tau\):

**Lemma 11** For any type \(\tau\), the reduction of any term in \(\mathcal{T}_\tau\) halts.
3.5. NORMALIZATION

Reduction of closed terms of type \( \tau \) preserves membership in \( \mathcal{T}_\tau \):

**Lemma 12** If \( \emptyset \vdash M : \tau \) and \( M \rightarrow M' \), then \( M \in \mathcal{T}_\tau \) iff \( M' \in \mathcal{T}_\tau \).

Therefore, it just remains to show that any term of type \( \tau \) is in \( \mathcal{T}_\tau \):

**Lemma 13** If \( \emptyset \vdash M : \tau \), then \( M \in \mathcal{T}_\tau \).

The proof is by induction on (the typing derivation of) \( M \). However, the case for abstraction requires some similar statement, but for open terms. We need to strengthen the lemma. Actually, to avoid considering open terms, we instead require the statement to hold for all closed instances of an open term:

**Lemma 14 (strengthened)** If \( (x_i : \tau_i)^{i\in I} \vdash M : \tau \), then for any closed values \( (V_i)^{i\in I} \) in \( (\mathcal{T}_{\tau_i})^{i\in I} \), the term \( [(x_i \mapsto V_i)^{i\in I}]M \) is in \( \mathcal{T}_\tau \).

---

**Proof:** We write \( \Gamma \) for \( (x_i : \tau_i)^{i\in I} \) and \( \theta \) for \( [(x_i \mapsto V_i)^{i\in I}] \). Assume \( \Gamma \vdash M : \tau \) (1) and \( (V_i)^{i\in I} \) in \( (\mathcal{T}_{\tau_i})^{i\in I} \) (2). We show that \( \theta M \) is in \( \mathcal{T}_\tau \) (3) by structural induction on \( M \).

**Case \( M \) is \( x_i \):** Immediate since the conclusion (3) is one of the hypotheses (2).

**Case \( M \) is \( M_1 \cdot M_2 \):** By inversion of the typing judgment (1), we have \( \Gamma \vdash M_1 : \tau_2 \rightarrow \tau \) (4) and \( \Gamma \vdash M_2 : \tau_2 \) (5) for some type \( \tau_2 \). Therefore, by induction hypothesis applied to (4) and (5), we have \( \theta M_1 \in \mathcal{T}_{\tau_2 \rightarrow \tau} \) and \( \theta M_2 \in \mathcal{T}_{\tau_2} \). Thus, by definition of \( \mathcal{T}_\tau \), we have \( \theta M_1 \cdot \theta M_2 \in \mathcal{T}_\tau \); that is, \( \theta M \in \mathcal{T}_\tau \).

**Case \( M \) is \( \lambda x : \tau_1 . M_2 \):** By inversion of the typing judgment (1), we have \( \Gamma, x : \tau_1 \vdash M : \tau_2 \) (6) where \( \tau_1 \rightarrow \tau_2 \) is \( \tau \) (7). Since \( M \) is a value, it is terminating. Hence, to ensure (3), it suffices to show that the application of \( \theta M \) to any \( M_1 \) in \( \mathcal{T}_{\tau_1} \) is in \( \mathcal{T}_{\tau_2} \) (8). Let \( M_1 \in \mathcal{T}_{\tau_1} \). By definition of \( \mathcal{T}_{\tau_1} \), the term \( M_1 \) reduces to some value \( V \), which by subject reduction has type \( \tau_1 \), and so is in \( \mathcal{T}_{\tau_1} \) (9). We have:

\[
(\theta M) M_1 \triangleq (\theta (\lambda x : \tau_1 . M_2)) M_1 \quad \text{by definition of } M
\]

\[
= (\lambda x : \tau_1 . \theta M_2) M_1 \quad \text{choose } x \neq \bar{x}
\]

\[
\rightarrow^\ast (\lambda x : \tau_1 . \theta M_2) V \quad \text{by (9)}
\]

\[
\rightarrow [x \mapsto V]\theta M_2 \quad \text{by (\beta)}
\]

\[
\in \mathcal{T}_{\tau_2} \quad \text{by I.H.}
\]

In the last step, we may apply the induction hypothesis, since the first hypothesis is (6) and the second one follows from (2) and (9). In summary, \( (\theta M) M_1 \) reduces to a term in \( \mathcal{T}_{\tau_2} \). Since \( \mathcal{T}_{\tau_2} \) is closed by reduction, \( (\theta M) M_1 \) itself is in \( \mathcal{T}_{\tau_2} \), which establishes (8), as expected.
3.6 Simple extensions

In this section, we introduce simple extensions to the calculus, mainly adding new constants and new primitives. These extensions will look very similar in one another and we will see how they can be factored out in the case of System F.

3.6.1 Unit

This is one of the simplest extension. We just introduce a new type `unit` and a constant value `()` of that type.

\[
\tau ::= \ldots | \text{unit} \hspace{1cm} V ::= \ldots | () \hspace{1cm} M ::= \ldots | ()
\]

Reduction rules are unchanged, since `()` is already a value. The following typing rule is introduced:

\[
\text{UNIT} \hspace{1cm} \Gamma \vdash () : \text{unit}
\]

**Exercise 13** Check that type soundness is preserved.  
(Solution p. [40])

Notice that the classification Lemma is no more degenerate.

3.6.2 Boolean

\[
V ::= \ldots | \text{true} | \text{false} \hspace{1cm} M ::= \ldots | \text{true} | \text{false} | \text{if } M \text{ then } M \text{ else } M
\]

We add only one evaluation context, since only the condition should be reduced:

\[
E ::= \ldots | \text{if } [] \text{ then } M \text{ else } M
\]

In particular, if \( V \) then \( E \) else \( M \) or if \( V \) then \( E \) else \( M \) are not evaluation contexts, because \( M \) and \( N \) must not be both evaluated before the conditional has been resolved. Instead, once the condition is a value, the conditional can be reduced to the relevant branch and dropping the other one, by one of the two new reduction rules:

\[
\text{if true then } M_1 \text{ else } M_2 \rightarrow M_1 \hspace{1cm} \text{if false then } M_1 \text{ else } M_2 \rightarrow M_2
\]

We also introduction a new type, `bool`, to classify booleans.

\[
\tau ::= \ldots | \text{bool}
\]

The new typing rules are:

\[
\begin{array}{c}
\text{TRUE} \\
\Gamma \vdash \text{true} : \text{bool}
\end{array} \hspace{1cm} \begin{array}{c}
\text{FALSE} \\
\Gamma \vdash \text{false} : \text{bool}
\end{array} \hspace{1cm} \begin{array}{c}
\text{IFTHENELSE} \\
\Gamma \vdash M_0 : \text{bool} \hspace{1cm} \Gamma \vdash M_1 : \tau \hspace{1cm} \Gamma \vdash M_2 : \tau \hspace{1cm} \Gamma \vdash \text{if } M_0 \text{ then } M_1 \text{ else } M_2 : \tau
\end{array}
\]
Exercise 14 Given the new cases for the classification lemma (without proving them). Check that progress is preserved. (Solution p. 46)

Exercise 15 Describe the extension of the \( \lambda \)-calculus with integers addition, and multiplication. (We do not ask to recheck the meta-theory, just to give the changes to the syntax and static and dynamic semantics, as we did above for booleans.) (Solution p. 46)

### 3.6.3 Pairs

To extend the simply-typed \( \lambda \)-calculus with pairs, we extend values, expressions, and evaluation contexts as follows:

\[
\begin{align*}
\text{i} & ::= 1 | 2 \\
\text{M} & ::= \ldots | (\text{M}, \text{M}) | \text{proj}_i \text{M} \\
\text{V} & ::= \ldots | (\text{V}, \text{V}) \\
\text{E} & ::= \ldots | ([], \text{M}) | (\text{V}, []) | \text{proj}_i []
\end{align*}
\]

Notice that the components of the pair are evaluated from left-to-right. At this stage, it could be left unspecified as the language is pure. However, it should be fixed when we later extend the language with side effects—even if the user should avoid side effects during evaluation of the components of a pair. This orientation from left-to-right is somewhat arbitrary—but more intuitive than the opposite order!

We introduce one new reduction rule (in fact, two rules if we inlined \( i \)):

\[
\text{proj}_i (\text{V}_1, \text{V}_2) \rightarrow \text{V}_i
\]

Product types are introduced to classify pairs, together with two new typing rules:

\[
\frac{\text{PAIR}}{
\Gamma \vdash \text{M}_1 : \tau_1 \\
\Gamma \vdash \text{M}_2 : \tau_2}
\quad
\frac{\text{PROJ}}{
\Gamma \vdash \text{M} : \tau_1 \times \tau_2}
\]

\[
\quad
\frac{
\Gamma \vdash \text{M}_1 : \tau \\
\Gamma \vdash \text{M}_2 : \tau}{
\Gamma \vdash (\text{M}_1, \text{M}_2) : \tau \\
\Gamma \vdash \text{proj}_i \text{M} : \tau_i}
\]

Exercise 16 Check that subject reduction is preserved when adding pairs. (Solution p. 47)

Exercise 17 Modify the semantics to evaluate pairs from right to left. Would this be sound? Would this be still call-by-value? (Solution p. 47)

### 3.6.4 Sums

Values, expressions, evaluation contexts are extended:

\[
\begin{align*}
\text{M} & ::= \ldots | \text{inj}_i \text{M} | \text{case} \text{M} \text{ of } \text{V} \circ \text{V} \\
\text{V} & ::= \ldots | \text{inj}_i \text{V} \\
\text{E} & ::= \ldots | \text{inj}_i [] | \text{case} [] \text{ of } \text{V} \circ \text{V}
\end{align*}
\]

A new reduction rule is introduced:

\[
\text{case} \text{inj}_i \text{V} \text{ of } V_1 \circ V_2 \rightarrow V_i V
\]
Sum types are added to classify sums:

\[ \tau ::= \ldots | \tau + \tau \]

Two new typing rules are introduced:

\[
\frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \text{inj}_i M : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash M : \tau_1 + \tau_2 \quad \Gamma \vdash V_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash V_2 : \tau_2 \rightarrow \tau}{\Gamma \vdash \text{case } M \text{ of } V_1 \diamond V_2 : \tau}
\]

Notice  A property of the simply-typed \( \lambda \)-calculus is lost: expressions do not have unique types anymore, i.e. the type of an expression is no longer always determined by the expression. Uniqueness of types may however be recovered by using a type annotation in injections:

\[ V ::= \ldots | \text{inj}_i V \text{ as } \tau \]

and modifying the typing rules and reduction rules accordingly. Although, the later variant is more verbose (and so not chosen in practice) it is easier and thus usually the one chosen for meta-theoretical studies.

Exercise 18  Describe the extension with the option type.

3.6.5 Modularity of extensions

The three preceding extensions are very similar. Each one introduces:

- a new type constructor, to classify values of a new shape;
- new expressions, to construct and destruct values of a new shape.
- new typing rules for new forms of expressions;
- new reduction rules, to specify how values of the new shape can be destructed;
- new evaluation contexts, but just to propagate reduction under the new constructors.

Then, in each case,

- subject reduction is preserved because types of new redexes are preserved by the new reduction rules.
- progress is preserved because the type system ensures that the new destructors can only be applied to values such that at least one of the new reduction rules applies.
Moreover, the extensions are independent: they can be added to the \( \lambda \)-calculus alone or mixed altogether. Indeed, no assumption about other extensions (the “…” ) has ever been made, except for the classification lemma which requires, informally, that \textit{values of other shapes have types of other shapes}. This is obviously the case in the extensions we have presented: the unit has the unit type, pairs have product types, and sums have sum types.

In fact, all these extensions could have been presented as several instances of a more general extension of the \( \lambda \)-calculus with constants, for which type soundness can be established uniformly under reasonable assumptions relating the typing rules and reduction rules for constants. This is the approach that we will follow in the next chapter (§4).

### 3.6.6 Recursive functions

Programs in the simply-typed \( \lambda \)-calculus always terminate. In particular, fix points of the \( \lambda \)-calculus cannot be typed. To recover recursion, we may introduce recursive functions as follows. Values and expressions are extended with a fix-point construct:

\[
\begin{align*}
V := & \ldots | \mu f:\tau.\lambda x.M \\
M := & \ldots | \mu f:\tau.\lambda x.M
\end{align*}
\]

A new reduction rule is introduced to unfold recursive calls:

\[
(\mu f:\tau.\lambda x.M) \ V \xrightarrow{\text{fold}} [f \mapsto \mu f:\tau.\lambda x.M] [x \mapsto V] M
\]

Types are \textit{not} extended, as we already have function types, \textit{i.e.} types won’t tell the difference between a function and a recursive function. A new typing rule is introduced:

\[
\begin{align*}
\text{FixAbs} \\
\Gamma, f : \tau_1 \rightarrow \tau_2 \vdash \lambda x: \tau_1. M : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash \mu f : \tau_1 \rightarrow \tau_2. \lambda x. M : \tau_1 \rightarrow \tau_2
\end{align*}
\]

In the premise, the type \( \tau_1 \rightarrow \tau_2 \) serves as both an assumption and a goal. This is a typical feature of recursive definitions.

Notice that we have syntactically restricted recursive definitions to functions. We could allow the definition of recursive values as well. However, the definition of recursive expressions that are not syntactically values is more difficult, as their semantics may be undefined and their efficient compilation is problematic—no good solution has been found yet.

### 3.6.7 A derived construct: let-bindings

The let-binding construct \("let x : \tau = M_1 in M_2"\) can be viewed as syntactic sugar for the \( \beta \)-redex \("(\lambda x : \tau. M_2) M_1"\). The latter form can be type-checked \textit{only} by a derivation of the following shape:

\[
\begin{align*}
\text{Abs} \\
\Gamma, x : \tau_1 \vdash M_2 : \tau_2 \\
\Gamma \vdash \lambda x : \tau_1. M_2 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash M_1 : \tau_1 \\
\text{App} \\
\Gamma \vdash (\lambda x : \tau_1. M_2) M_1 : \tau_2
\end{align*}
\]
This means that the following derived rule is sound and complete for let-bindings:

\[
\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = M_1 \text{ in } M_2 : \tau_2}
\]

In the derived form let \( x : \tau_1 = M_1 \) in \( M_2 \) the type of \( M_1 \) must be given explicitly, although by uniqueness of types, it is fully determined by the expression \( M_1 \) and is thus redundant. If we replace the derived form by a primitive form let \( x = M_1 \) in \( M_2 \) we could use the following primitive typing rule.

\[
\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}
\]

**Remark 3** The primitive form is not necessary a better design choice however. Derived forms are more economical, since they do not extend the core language, and should be used whenever possible. Minimizing the number of language constructs is at least as important as avoiding extra type annotations in an explicitly-typed language. Moreover, removing redundant type annotations is the problem of type reconstruction and we should not bother too much about it in the explicitly-typed version of the language.

**Sequences** The sequence “\( M_1; M_2 \)” is a derived construct of let-bindings; it can be viewed as additional syntactic sugar that expands to let \( x : \text{unit} = M_1 \) in \( M_2 \) where \( x \neq M_2 \).

**Exercise 19** Recover the typing rule for sequences from this syntactic sugar.

**A derived construct: let rec** The construct “let rec \(( f : \tau) x = M_1 \) in \( M_2 \)” can also be viewed as syntactic sugar for “let \( f = \mu f : \tau. \lambda x. M_1 \) in \( M_2 \)” . The latter can be type-checked only by a derivation of the form:

\[
\frac{\Gamma, f : \tau \rightarrow \tau_1; x : \tau \vdash M_1 : \tau_1 \quad \Gamma, f : \tau \rightarrow \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let rec } f = \mu f : \tau \rightarrow \tau_2. \lambda x. M_1 \text{ in } M_2 : \tau_2}
\]

This means that the following derived rule is sound and complete:

\[
\frac{\Gamma, f : \tau \rightarrow \tau_1; x : \tau \vdash M_1 : \tau_1 \quad \Gamma, f : \tau \rightarrow \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let rec } \left( f : \tau \rightarrow \tau_1 \right) x = M_1 \text{ in } M_2 : \tau_2}
\]

**3.7 Exceptions**

Exceptions are a mechanism for changing the normal order of evaluation (usually, but not necessarily, in case something abnormal occurred).
When an exception is raised, the evaluation does not continue as usual: Shortcutting normal evaluation rules, the exception is propagated up into the evaluation context until some handler is found at which the evaluation resumes with the exceptional value received; if no handler is found, the exception reaches the toplevel and the result of the evaluation is the exception instead of a value.

Because exceptions may break the flow of evaluation, they cannot be described as just new constants and primitives.

3.7.1 Semantics

We extend the language with a constructor form to raise an exception and a destructor form to catch an exception; we also extend the evaluation contexts:

\[
M ::= \ldots | \text{raise } M | \text{try } M \text{ with } M \\
E ::= \ldots | \text{raise } [] | \text{try } [] \text{ with } M
\]

However, we do not treat `raise V` as a value, since `raise V` stops the normal order of evaluation. Instead, we introduce three reduction rules to propagate and handle exceptions:

\[
\text{Raise} \quad \text{Handle-Val} \quad \text{Handle-Raise}
\]
\[
F[\text{raise } V] \rightarrow \text{raise } V \\
\text{try } V \text{ with } M \rightarrow V \\
\text{try raise } V \text{ with } M \rightarrow M V
\]

Rule `Raise` propagates an exception one level up in the evaluation contexts, but not through a handler. This is why the rule uses an evaluation context \( F \), which stands for any evaluation context \( E \) other than `try [] with M`.

The handling of exceptions is then treated by two specific rules: Rule `Handle-Raise` passes an exceptional value to its handler; Rule `Handle-Val` removes the handler around a value.

Example  Assume that \( K \) is \( \lambda x. \lambda y. y \) and \( M \rightarrow V \). We have the following reduction:

\[
\begin{align*}
\text{try } K (\text{raise } M) \text{ with } \lambda x. x & \quad \text{by Context} \\
\rightarrow \text{try } K (\text{raise } V) \text{ with } \lambda x. x & \quad \text{by Raise} \\
\rightarrow \text{try raise } V \text{ with } \lambda x. x & \quad \text{by Handle-Raise} \\
\rightarrow (\lambda x. x) V & \quad \text{by } \beta \\
\rightarrow V &
\end{align*}
\]

In particular, we do not have the following reduction sequence, since `raise V` is not a value, hence the \( K (\text{raise } V) \) does not reduce to \( \lambda y. y \):

\[
\text{try } K (\text{raise } V) \text{ with } \lambda x. x \rightarrow \rightarrow \text{try } \lambda y. y \text{ with } \lambda x. x \rightarrow \lambda y. y
\]
3.7.2 Typing rules

We assume given a fixed type \texttt{exn} for exceptional values. The new typing rules are:

\[
\begin{align*}
\text{Raise} & : \quad \Gamma \vdash M : \texttt{exn} \\
\Gamma & \vdash \text{raise } M : \tau \\
\text{Try} & : \quad \Gamma \vdash M_1 : \tau \\
& \quad \Gamma \vdash M_2 : \texttt{exn} \rightarrow \tau \\
& \quad \Gamma \vdash \text{try } M_1 \text{ with } M_2 : \tau
\end{align*}
\]

There are some subtleties: \texttt{raise} turns an expression of type \texttt{exn} into an exception. Consistently, the handler has type \texttt{exn} \rightarrow \tau, since it receives as argument the value of type \texttt{exn} that has been raised. The expression \texttt{raise } M can have any type, since the current computation is aborted. In \texttt{try } M_1 \text{ with } M_2, M_2 must return a value of the same type as \texttt{M}_1, since the evaluation will proceed with either branch depending on whether the evaluation of \texttt{M}_1 raises an exception or returns a value.

Type of exceptions What can we choose for \texttt{exn}? Well, any type could do. Choosing \texttt{unit}, exceptions would carry no information. Choosing \texttt{int}, exceptions would carry an integer that could be used, \textit{e.g.}, to report some error code. Choosing \texttt{string}, exceptions would carry a string that could be used to report error messages. Or better, exception could be of a sum type to allow any of these alternatives to be chosen when the exception is raised.

This is the approach followed by ML. However, since the set of exceptions is not known in advance, ML declares a new type \texttt{exn} for exceptions and allows adding new cases to the sum later on as needed. This is called an extensible datatype. As a counterpart checking for exceptions can’t be exhaustive without a “catch all” branch, since further cases could always be added later. Notice that although new constructors may be added, the type of exception is fixed in the whole program, to \texttt{exn}. This is essential for type soundness, since the handling and raising of exceptions must agree globally on the type \texttt{exn} of exceptional values as it is not passed around.

Notice that exception constructors must have closed types since the type \texttt{exn} has no parameter.

Type soundness How do we state type soundness, since exceptions may be uncaught? By saying that this is the only “exception” to progress:

\textbf{Theorem 4 (Progress)} A well-typed, irreducible term is either a value or an uncaught exception. if \( \emptyset \vdash M : \tau \) and \( M \rightarrow \), then \( M \) is either \( v \) or \texttt{raise } v for some value \( v \).

\textbf{Exercise 20} Do all well-typed closed programs still terminate in the presence of exceptions?

(Solution p. [42])

3.7.3 Variations

Structured exceptions We have assumed that there is a unique exception, which could itself be a sum type. This simulates having multiple exceptions where each one is identified
by a tag and may carry values of different types. However, having multiple exceptions as primitive would amount to redefining sum types within the mechanism of exceptions; this would just bringing more complications without any real gain.

**On uncaught exceptions** Usage of exceptions may vary a lot in programs: some exceptions are used for fatal errors and abort the program while others may be used during normal computation, *e.g.* for quickly returning from a deep recursive call. However, an uncaught exception is often a programming error—even exceptions raised to abort the whole program must usually be caught for error reporting or cleaning up before exiting. It may be surprising that uncaught exceptions are not considered as static errors that should be detected by the type system.

Unfortunately, detecting uncaught exceptions require more expressive type systems and the existing solutions are often complicated for some limited benefit. This explains why they are not often used in practice.

The complication comes from the treatment of functions, which have some *latent effect* of possibly raising or catching an exception when applied. To be precise, the analysis must therefore enrich types of functions with latent effects, which is quite invasive and obfuscating.

Uncaught exceptions are checked in the language Java, but they must be declared. See [Leroy and Pessaux (2000)](#) for an analysis of uncaught exceptions in ML.

**Small variation** Once raised, exceptions are propagated step-by-step by Rule `Raise` until they reach a handler or the toplevel. The semantics could avoid the step-by-step propagation of exceptions by handling exceptions deeply inside terms. It suffices to replace the three reduction rules by:

\[
\begin{align*}
\text{HANDLE-VAL'} & : \text{try } V \text{ with } M \rightarrow V \\
\text{HANDLE-RAISE'} & : \text{try } F \left[ \text{raise } V \right] \text{ with } M \rightarrow M \, V
\end{align*}
\]

where \( F \) is sequence of \( F \)-contexts, *i.e.* a handler-free evaluation context of arbitrary depth. In this case, uncaught exceptions are of the form \( F[V] \). This semantics is perhaps more intuitive—but it is equivalent.

**Exceptions with bindings** [Benton and Kennedy (2001)](#) have argued for merging let-bindings with exception handling into a unique form `let x = M_1 with M_2 in M_3`. The expression \( M_1 \) is evaluated first and, if it returns a value, it is substituted for \( x \) in \( M_3 \), as if we had evaluated `let x = M_1 in M_3`; otherwise, *i.e.*, if it raises an exception `raise V`, then the exception is handled by \( M_2 \), as if we had evaluated `try M_1 with M_2`.

This combined form captures a common pattern in programming that has no elegant workaround:

\[
\begin{align*}
\text{let rec } \text{read_config_in_path} \text{ filename (dir :: dirs)} \rightarrow \\
\text{let } \text{fd} = \text{open_in} \left( \text{Filename.concat dir filename} \right)
\end{align*}
\]
### CHAPTER 3. SIMPLY-TYPED LAMBDA-CALCULUS

with \( \text{Sys\_error} \rightarrow \text{read\_config\_filename\_dirs\_in} \)

\[ \text{read\_config\_from\_fd\_fd} \]

This form is also better suited for program transformations, as argued by [Benton and Kennedy (2001)].

The separate let-binding and exception handling constructs are obviously particular cases of the new combined construct. Conversely, encoding the new construct `let x = M_1 with M_2 in M_3` with `let` and `try` is not so easy. In particular, it is not equivalent to: `try (let x = M_1 in M_3) with M_2!` In this expression, \( M_3 \) could raise an exception that would then be handled by \( M_2 \), which is not intended.

There are several encodings in the combined form into simple exceptions, but none of them is very readable, and all of them introduce some source of inefficiency. For instance, one may use a sum datatype to tell whether \( M_1 \) raised an exception:

\[
\text{case (try Val M}_1\text{ with }\lambda y.\text{Exc } y\text{) of (Val: }\lambda x.\text{M}_3\text{ o Exc: }\text{M}_2)\]

Alternatively, one may freeze the continuation \( M_3 \) while handling the exception:

\[
(\text{try let } x = M_1 \text{ in } \lambda () . M_3 \text{ with } \lambda y.\lambda () . M_2 y) ()
\]

The extra allocation for the sum or the closure for the continuation are sources of inefficiency which the primitive combined form can easily avoid.

**Exercise 21** Describes the dynamic semantics of the `let x = M_1 with M_2 in M_3` construct, formally.  
(Solution p. 48)

**Exercise 22** (try finalize) A finalizer is some code that should be run in case of both normal and exceptional evaluation. Write a function `finalize` that takes four arguments \( f, x, g, \) and \( y \) and returns the application \( f x \) with finalizing code \( g y \). i.e. \( g y \) should be called before returning the result of the application of \( f \) to \( x \) whether it executes normally or raises an exception.  
(Solution p. 48)

### 3.8 References

In the ML vocabulary, a *reference cell*, also called a *reference*, is a dynamically allocated block of memory that holds a value and whose content can change over time. A reference can be allocated and initialized \( (\text{ref}) \), written \( (: =) \), and read \( (!) \). Expressions and evaluation contexts are extended as follows:

\[
M ::= \ldots | \text{ref } M | M := M | ! M \\
E ::= \ldots | \text{ref } [] | [] := M | V := [] | ! []
\]

A reference allocation expression is not a value. Otherwise, by \( \beta \)-reduction, the program:

\[
(\lambda x: \tau. (x := 1 ; ! x)) \ (\text{ref } 3)
\]
which intuitively should yield 1, would reduce to:

\[(\text{ref } 3) := 1; ! (\text{ref } 3)\]

which intuitively yields 3. How shall we solve this problem? The expression \((\text{ref } 3)\) should first reduce to a value: the address of a fresh cell. That is, not just the content of a cell matters, but also its address, since writing through one copy of the address should not affect a future read via another copy.

### 3.8.1 Language definition

Formally, we extend the simply-typed \(\lambda\)-calculus with memory locations:

\[
\begin{align*}
M & ::= \ldots | \ell \\
V & ::= f \ldots | \ell
\end{align*}
\]

A memory location is just an atom (that is, a name). The value found at a location \(\ell\) is obtained by indirection through a memory (or store). A memory \(\mu\) is a finite mapping of locations to closed values. A configuration is a pair \(M/\mu\) of a term and a store. The operational semantics (given next) reduces configurations instead of expressions.

The semantics maintains a no-dangling-pointers invariant: the locations that appear in \(M\) or in the image of \(\mu\) are in the domain of \(\mu\). Initially, the store is empty, and the term contains no locations, because, by convention, memory locations cannot appear in source programs. So, the invariant holds.

If we wish to start reduction with a non-empty store, we must check that the initial configuration satisfies the no-dangling-pointers invariant. Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

\[
\begin{align*}
(\lambda x: \tau. M) V /\mu & \rightarrow [x \mapsto V] M /\mu \\
E[M] /\mu & \rightarrow E[M'] /\mu' & \text{if } M /\mu \rightarrow M' /\mu'
\end{align*}
\]

Three new reduction rules are added:

\[
\begin{align*}
\text{ref } V /\mu & \rightarrow \ell /\mu[\ell \mapsto V] & \text{if } \ell \notin \text{dom}(\mu) \\
\ell ::= V /\mu & \rightarrow \ell /\mu[\ell \mapsto V] \\
! \ell /\mu & \rightarrow \mu(\ell) /\mu
\end{align*}
\]

In the last two rules, the no-dangling-pointers invariant guarantees \(\ell \in \text{dom}(\mu)\).

The type system is modified as follows. Types are extended:

\[
\tau ::= \ldots | \text{ref } \tau
\]

Three new typing rules are introduced:

\[
\begin{align*}
\text{Ref} & \quad \Gamma \vdash M : \tau \\
\text{Set} & \quad \Gamma \vdash M_1 : \text{ref } \tau \quad \Gamma \vdash M_2 : \tau \\
\text{Get} & \quad \Gamma \vdash M : \text{ref } \tau
\end{align*}
\]

\[
\Gamma \vdash \text{ref } M : \text{ref } \tau
\]

\[
\Gamma \vdash M_1 := M_2 : \text{unit}
\]

\[
\Gamma \vdash ! M : \tau
\]

Is that all we need? The preceding setup is enough to typecheck source terms, but does not
allow stating or proving type soundness. Indeed, we have not yet answered these questions: What is the type of a memory location $\ell$? When is a configuration $M / \mu$ well-typed? A location $\ell$ has type \texttt{ref $\tau$ when it points to some value of type $\tau$}. Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \texttt{ref $\tau$}}$$

Then, typing judgments would have the form $\mu, \Gamma \vdash M : \tau$. Typing judgments would no longer be \textit{inductively} defined (or else, every cyclic structure would be ill-typed). Instead, co-\textit{induction} would be required. Moreover, if the value $\mu(\ell)$ happens to admit two distinct types $\tau_1$ and $\tau_2$, then $\ell$ admits types \texttt{ref $\tau_1$} and \texttt{ref $\tau_2$}. So, one can write at type $\tau_1$ and read at type $\tau_2$: this rule is \textit{unsound}! A simpler, and sound, approach is to fix the type of a memory location when it is first allocated. To do so, we use a \textit{store typing} $\Sigma$, a finite mapping of locations to types.

Then, a location $\ell$ has type \texttt{ref $\tau$ “when the store typing $\Sigma$ says so.”}

$$\frac{}{\Sigma, \Gamma \vdash \ell : \texttt{ref $\Sigma(\ell)$}}$$

Typing judgments now have the form $\Sigma, \Gamma \vdash M : \tau$. The following typing rules for stores and configurations ensure that the store typing predicts appropriate types

$$\frac{\forall \ell \in \text{dom}($\mu$), \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma}$$

$$\frac{\Sigma, \varnothing \vdash M : \tau \vdash \mu : \Sigma}{\vdash M / \mu : \tau}$$

Remarks:

- This is an \textit{inductive} definition. The store typing $\Sigma$ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.

- The store typing is used only in the definition of a “well-typed configuration” and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.

### 3.8.2 Type soundness

The type soundness statements are slightly modified in the presence of the store, since we now reduce configurations:

**Theorem 5 (Subject reduction)** Reduction preserves types: if $M / \mu \rightarrow M' / \mu'$ and $\vdash M / \mu : \tau$, then $\vdash M' / \mu' : \tau$.

**Theorem 6 (Progress)** If $M / \mu$ is a well-typed, irreducible configuration, then $M$ is a value.
Inlining subject reduction can also be restated as:

**Theorem 7 (Subject reduction, expanded)** If \( M / \mu \rightarrow M' / \mu' \) and \( \Sigma, \emptyset \vdash M : \tau \) and \( \vdash \mu : \Sigma \), then there exists \( \Sigma' \) such that \( \Sigma', \emptyset \vdash M' : \tau \) and \( \vdash \mu' : \Sigma' \).

This statement is correct, but *too weak*—its proof by induction will fail in one case. Let us look at the case of reduction under a context. The hypotheses are:

\[
M / \mu \rightarrow M' / \mu' \quad \text{and} \quad \Sigma, \emptyset \vdash E[M] : \tau \quad \text{and} \quad \vdash \mu : \Sigma
\]

Assuming compositionality, there exists \( \tau' \) such that:

\[
\Sigma, \emptyset \vdash M : \tau' \quad \text{and} \quad M', \ (\Sigma, \emptyset \vdash M' : \tau') \Rightarrow (\Sigma, \emptyset \vdash E[M'] : \tau)
\]

Then, by the induction hypothesis, there exists \( \Sigma' \) such that:

\[
\Sigma', \emptyset \vdash M' : \tau' \quad \text{and} \quad \vdash \mu' : \Sigma'
\]

Here, we are stuck. The context \( E \) is well-typed under \( \Sigma \), but the term \( M' \) is well-typed under \( \Sigma' \), so we cannot combine them. We are missing a key property: the store typing grows with time. That is, although new memory locations can be allocated, the type of an existing location does not change. This is formalized by strengthening the subject reduction statement:

**Theorem 8 (Subject reduction, strengthened)** If \( M / \mu \rightarrow M' / \mu' \) and \( \Sigma, \emptyset \vdash M : \tau \) and \( \vdash \mu : \Sigma \), then there exists \( \Sigma' \) such that \( \Sigma', \emptyset \vdash M' : \tau \) and \( \vdash \mu' : \Sigma' \) and \( \Sigma \subseteq \Sigma' \).

At each reduction step, the new store typing \( \Sigma' \) extends the previous store typing \( \Sigma \). Growing the store typing preserves well-typedness (a generalization of the weakening lemma):

**Lemma 15 (Stability under memory allocation)** If \( \Sigma \subseteq \Sigma' \) and \( \Sigma, \Gamma \vdash M : \tau \), then \( \Sigma', \Gamma \vdash M : \tau \).

This allows establishing a strengthened version of compositionality:

**Lemma 16 (Compositionality)** Assume \( \Sigma, \emptyset \vdash E[M] : \tau \). Then, there exists \( \tau' \) such that:

- \( \Sigma, \emptyset \vdash M : \tau' \),
- for every \( \Sigma' \) and \( M' \), if \( \Sigma \subseteq \Sigma' \) and \( \Sigma', \emptyset \vdash M' : \tau' \), then \( \Sigma', \emptyset \vdash E[M'] : \tau \).

Let us now look again at the case of reduction under a context. The hypotheses are:

\[
\Sigma, \emptyset \vdash E[M] : \tau \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad M / \mu \rightarrow M' / \mu'
\]

By compositionality, there exists \( \tau' \) such that:

\[
\Sigma, \emptyset \vdash M : \tau'
\]

\[
\forall \Sigma', \forall M', \ (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \emptyset \vdash M' : \tau') \Rightarrow (\Sigma', \emptyset \vdash E[M'] : \tau')
\]
By the induction hypothesis, there exists $\Sigma'$ such that:

$$\Sigma', \varnothing \vdash M' : \tau' \quad \text{and} \quad \vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma \subseteq \Sigma'$$

The goal immediately follows.

**Exercise 23** Prove subject reduction and progress for simply-typed $\lambda$-calculus equipped with unit, pairs, sums, recursive functions, exceptions, and references.

### 3.8.3 Tracing effects with a monad

Haskell adopts a different route and chooses to distinguish effectful computations (Peyton Jones and Wadler, 1993; Peyton Jones, 2009).

- **return**: $\alpha \to \text{IO} \alpha$
- **bind**: $\text{IO} \alpha \to (\alpha \to \text{IO} \beta) \to \text{IO} \beta$
- **main**: $\text{IO} ()$
- **newIORef**: $\alpha \to \text{IO} (\text{IORef} \alpha)$
- **readIORef**: $\text{IORef} \alpha \to \text{IO} \alpha$
- **writeIORef**: $\text{IORef} \alpha \to \alpha \to \text{IO} ()$

Haskell offers many monads other than IO. In particular, the ST monad offers references whose lifetime is statically controlled.

### 3.8.4 Memory deallocation

In ML, memory deallocation is implicit. It must be performed by the runtime system, possibly with the cooperation of the compiler. The most common technique is garbage collection. A more ambitious technique, implemented in the ML Kit, is compile-time region analysis (Tofte et al., 2004).

References in ML are easy to typecheck, thanks to the no-dangling-pointers property of the semantics. Making memory deallocation an explicit operation, while preserving type soundness, is possible, but difficult. This requires reasoning about aliasing and ownership. See Charguéraud and Pottier (2008) for citations. See Pottier and Protzenko (2013) for the language Mezzo designed especially for the explicit control of resources. The meta-theory of such languages may become quite intricate (Pottier, 2013).

### Further reading

For a textbook introduction to $\lambda$-calculus and simple types, see Pierce (2002). For more details about syntactic type soundness proofs, see Wright and Felleisen (1994).
3.9 Ommitted proofs and answers to exercises

Solution of Exercise 8

See the statement of bisimilation for System-F in §4.4.5 in particular lemmas 25 and ??.

Solution of Exercise 10

Case \( M = M_1 M_2 \): By inversion of the judgment \( \Gamma \vdash M : \tau \), we must have \( \Gamma \vdash M_1 : \tau_2 \rightarrow \tau \) and \( \Gamma \vdash M_2 : \tau_2 \) for some \( \tau_2 \). By induction hypothesis, we have \( \Gamma, y : \tau' \vdash M_1 : \tau_2 \rightarrow \tau \) and \( \Gamma, y : \tau' \vdash M_2 : \tau_2 \), respectively. We conclude by an application of Rule \textsc{App}.

Solution of Exercise 11

As a hint, the problem in the case for abstraction.

Solution of Exercise 12

\[
M \subseteq M' \iff \forall \Gamma, \forall \tau, (\Gamma \vdash M : \tau \implies \Gamma \vdash M' : \tau)
\]

Subject reduction can then be stated as \((\rightarrow) \subseteq (\subseteq)\). We proof it as follows:

Proof: Since \((\rightarrow)\) is the smallest relation that satisfies rules \textsc{Beta} and \textsc{Context}, it suffices to show that \((\subseteq)\) also satisfies rules \textsc{Beta} and \textsc{Context}.

Case \textsc{Beta}. Assume that \( \Gamma \vdash (\lambda x : \tau_0. M) V : \tau \). Then \( \Gamma \vdash [x \mapsto V]M : \tau \) follows by the substitution Lemma.

Case \textsc{Context}. Assume \( M \subseteq M' \). Let us show \( E[M] \subseteq E[M'] \). Assume \( \Gamma \vdash E[M] : \tau \). Then \( \Gamma \vdash E[M'] : \tau \) follows by compositinality.

Proof of Lemma 12

By induction on the structure of the type \( \tau \).

Case \( \tau \) is \( \alpha \): Then \( T_\alpha \) is the set of terms that terminates. If \( M \rightarrow M' \), the termination of \( M \), i.e. \( M \in T_\alpha \), is equivalent to the termination of \( M' \), i.e. \( M' \in T_\alpha \).

Case \( \tau \) is \( \tau_1 \rightarrow \tau_2 \): Then \( T_\tau \) is the set of terms of type \( \tau \) that terminate and also terminate when applied to any term \( M_1 \) of type \( \tau_1 \). Assume \( \emptyset \vdash M : \tau \) (1) and \( M \rightarrow M' \) (2). By subject reduction, we have \( \emptyset \vdash M' : \tau \). Moreover, from (2), termination of \( M \) and termination of \( M' \) are equivalent. Therefore, it only remains to check that for any term \( M_1 \) of \( T_{\tau_1} \), \( M M_1 \) and \( M' M_1 \) are both in \( T_{\tau_2} \) or both outside of \( T_{\tau_2} \) (3). Let \( M_1 \) be in \( T_{\tau_1} \). We have \( \emptyset \vdash M_1 : \tau \).
and thus $\emptyset \vdash M M_1 : \tau_2$. We also have the call-by-value reduction $M M_1 \rightarrow M' M_1$, Hence, (3) follows by induction hypothesis.

**Solution of Exercise 13**

Formally, we must revisit all the proofs. Auxiliary lemmas such as permutation and weakening still hold without any problem: in the proof by structural induction, there is a new case for unit expressions, which is proved by an application of the same rule, \texttt{UNIT} but with possibly a different context $\Gamma$.

In the proof of subject reduction, nothing need to be changed.

In the proof of progress, we have a new case for closed expressions, i.e. $()$, which happens to be a value, so it trivially satisfied the goal. Notice that although we do not need to invoke the classification for the new case of the $()$ expression, we still need to recheck the classification lemma, which is used in the case for application. The proof of the classification lemma is achieved by filling in the dots with a new case for a value of type \texttt{unit} that must be $()$, so that the classification can still be inverted.

**Solution of Exercise 14**

The new case for the classification Lemma is that a value of type \texttt{bool} must be a boolean, i.e. either \texttt{true} or \texttt{false} (4).

For the proof of progress, we assume that $\emptyset \vdash M : \tau$ (5) and show that $M$ is either a value or reducible (6) by structural induction on $M$. We have two new cases:

- **Case $M$ is \texttt{true} or \texttt{false}**: In both cases, $M$ is a value.
- **Case $M$ is \texttt{if} $M_0$ then $M_1$ else $M_2$**: By inversion of typing rules applied to (5), we have $\emptyset \vdash M_0 : \texttt{bool}$, $\emptyset \vdash M_1 : \tau$, and $\emptyset \vdash M_2 : \tau$. If $M_0$ is a value, then, since it is of type \texttt{bool}, it must be \texttt{true} or \texttt{false} by (4), and in both cases, $M$ reduces by either one of the two new rules. Otherwise, by induction hypothesis, $M_0$ must be reducible, and so is $M$ by rule \texttt{CONTEXT} since if $[]$ then $M_1$ else $M_2$ is an evaluation context. This ends the proof.

**Solution of Exercise 15**

This is very similar to the case of boolean, except that we introduce a denumerable collection of integer constants $(\bar{n})_{n \in \mathbb{N}}$.

$$V ::= \ldots | \bar{n} \quad M ::= \ldots | n \mid M + M \mid M \times M$$

We add only evaluation contexts:

$$E ::= \ldots | [] + M | V + [] | [] \times M | V \times []$$

two reduction rules are:

$$\bar{n} + \bar{m} \rightarrow n + m \quad \bar{n} \times \bar{m} \rightarrow n \times m$$
and the following typing rules:

\[
\begin{align*}
\text{\textsc{int}} & \quad \Gamma \vdash \bar{n} : \text{int} \\
\text{\textsc{plus}} & \quad \Gamma \vdash M_1 : \text{int} \quad \Gamma \vdash M_2 : \text{int} \\
& \quad \Gamma \vdash M_1 + M_2 : \text{int} \\
\text{\textsc{times}} & \quad \Gamma \vdash M_1 : \text{int} \quad \Gamma \vdash M_2 : \text{int} \\
& \quad \Gamma \vdash M_1 \times M_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{\textsc{proj}} & \quad \Gamma \vdash \text{proj}_i (V_1, V_2) \\
& \quad \Gamma \vdash V_1 : \tau_1 (1) \quad \Gamma \vdash V_2 : \tau_2 (2) \\
& \quad \Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2 (7)
\end{align*}
\]

\[
\begin{align*}
\text{\textsc{fold}} & = \lambda f : \tau \rightarrow \tau. \lambda x : \text{unit}. \text{let } z = \text{raise } f \text{ in } () \\
\text{\textsc{unfold}} & = \lambda f : \tau. \text{try let } z = f () \text{ in } \lambda x : \tau. x \text{ with } \lambda y : \tau \rightarrow \tau. y
\end{align*}
\]

\[
\begin{align*}
\text{\textsc{fold}} & \quad \text{\textsc{unfold}} \quad \lambda x. (\text{\textsc{unfold}} x) x \\
\text{\textsc{fold}} & \quad \text{\textsc{unfold}} \quad \omega (\text{\textsc{fold}} \omega)
\end{align*}
\]
Solution of Exercise 21

We need a new evaluation context:

\[
E ::= \ldots \mid \text{let } x = E \text{ with } M_2 \text{ in } M_3
\]

and the following reduction rules:

\begin{align*}
\text{Raise} & \quad F[\text{raise } V] \rightarrow \text{raise } V \\
\text{Handle-Val} & \quad \text{let } x = V \text{ with } M_2 \text{ in } M_3 \rightarrow [x \mapsto V]M_3 \\
\text{Handle-Raise} & \quad \text{let } x = \text{raise } V \text{ with } M_2 \text{ in } M_3 \rightarrow M_2 V
\end{align*}

Solution of Exercise 22

\begin{verbatim}
let finalize f x g y =
  let result = try f x with \text{exn} \rightarrow g y; raise \text{exn} in
  g y;
result
\end{verbatim}

An alternative that does not duplicate the finalizing code and could be inlined is:

\begin{verbatim}
type 'a result = Val of 'a | Exc of \text{exn}
let finalize f x g y =
  let result = try Val (f x) with \text{exn} \rightarrow Exc \text{exn} in
  g y;
  match result with Val x \rightarrow x | Exc \text{exn} \rightarrow raise \text{exn}
\end{verbatim}

As a counterpart, this allocated an intermediate result.
Chapter 4

Polymorphism and System F

4.1 Polymorphism

Polymorphism is the ability for a term to simultaneously admit several distinct types. Polymorphism is indispensable [Reynolds, 1974]: if a list-sorting function is independent of the type of the elements, then it should be directly applicable to lists of integers, lists of booleans, etc.. In short, it should have polymorphic type:

\[ \forall \alpha. (\alpha \to \alpha \to \text{bool}) \to \text{list } \alpha \to \text{list } \alpha \]

which can then be instantiated to any of the monomorphic types:

\[ (\text{int} \to \text{int} \to \text{bool}) \to \text{list int} \to \text{list int} \quad (\text{bool} \to \text{bool} \to \text{bool}) \to \text{list bool} \to \text{list bool} \quad \ldots \]

In the absence of polymorphism, the only ways of achieving this effect are either to manually duplicate the list-sorting function at every type (no-no!); or to use subtyping and claim that the function sorts lists of values of any type:

\[ (\top \to \top \to \text{bool}) \to \text{list } \top \to \text{list } \top \]

(The type \( \top \) is the type of all values, and the supertype of all types.) This leads to loss of information and subsequently requires introducing an unsafe downcast operation. This was the approach followed in Java before generics were introduced in 1.5.

Moreover, polymorphism seems to come almost for free, as it is already implicitly present in simply-typed \( \lambda \)-calculus. Indeed, all types of the compose functions are

\[ (\tau_1 \to \tau_2) \to (\tau_0 \to \tau_1) \to \tau_0 \to \tau_2 \]

among which is

\[ (\alpha_1 \to \alpha_2) \to (\alpha_0 \to \alpha_1) \to \alpha_0 \to \alpha_2 \]

which is principal, as all other types can be recovered by instantiation of the variables. By
saying that this term admits the polymorphic type
\[
\forall \alpha_1 \alpha_2 \cdot (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_0 \rightarrow \alpha_1) \rightarrow \alpha_0 \rightarrow \alpha_2
\]
we make polymorphism internal to the type system.

Polymorphism is a step on the road towards type abstraction. Intuitively, if a function that sorts a list has polymorphic type
\[
\forall \alpha \cdot (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list} \alpha \rightarrow \text{list} \alpha
\]
then it knows nothing about \( \alpha \)—it is parametric in \( \alpha \)—so it must manipulate the list elements abstractly: it can copy them around, pass them as arguments to the comparison function, but it cannot directly inspect their structure. In short, within the code of the list sorting function, the variable \( \alpha \) is an abstract type.

**Parametricity** In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it. For instance, the polymorphic type \( \forall \alpha \cdot \alpha \rightarrow \alpha \) has only one inhabitant, namely the identity. Similarly, the type of the list sorting function
\[
\forall \alpha \cdot (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list} \alpha \rightarrow \text{list} \alpha
\]
reveals a “free theorem” about its behavior! Basically, sorting commutes with (map \( f \)), provided \( f \) is order preserving. Note that there are many inhabitants of this type (e.g. a function that sorts in reverse order, or a function that removes duplicates) but they all satisfy this free theorem. This phenomenon was studied by Reynolds 1983 and by Wadler 1989; 2007, among others. An account based on an operational semantics is offered by Pitts 2000.

**Ad hoc versus parametric polymorphism** Let us begin a short digression. The term “polymorphism” dates back to a 1967 paper by Strachey (2000), where ad hoc polymorphism and parametric polymorphism were distinguished. There are two different (and sometimes incompatible) ways of defining this distinction:

- With parametric polymorphism, a term can admit several types, all of which are instances of a common polymorphic type: \( \text{int} \rightarrow \text{int}, \text{bool} \rightarrow \text{bool} \), \ldots and \( \forall \alpha \cdot \alpha \rightarrow \alpha \).

  With ad hoc polymorphism, a term can admit a collection of unrelated types: \( \text{int} \rightarrow \text{int} \rightarrow \text{int}, \text{float} \rightarrow \text{float} \rightarrow \text{float} \), \ldots but not \( \forall \alpha \cdot \alpha \rightarrow \alpha \rightarrow \alpha \).

- With parametric polymorphism, untyped programs have a well-defined semantics. (Think of the identity function.) Types are used only to rule out unsafe programs.

  With ad hoc polymorphism, untyped programs do not have a semantics: the meaning of a term can depend upon its type (e.g. \( 2 + 2 \)), or, even worse, upon its type derivation (e.g. \( \lambda x. \text{show} (\text{read} \, x) \)).
By the first definition, Haskell’s type classes (Hudak et al., 2007) are a form of (bounded) parametric polymorphism: terms have principal (qualified) type schemes, such as:

$$\forall \alpha. \text{Num}\, \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Yet, by the second definition, type classes are a form of ad hoc polymorphism: untyped programs do not have a semantics. This ends the digression.

### 4.2 Polymorphic \(\lambda\)-calculus

The System F, (also known as: the polymorphic \(\lambda\)-calculus; the second-order \(\lambda\)-calculus; \(F_2\)) was independently defined by Girard (1972) and Reynolds (1974).

#### 4.2.1 Types and typing rules

Types of the simply-typed \(\lambda\)-calculus are extended with polymorphic types:

$$\tau ::= \alpha \mid \tau \Rightarrow \tau \mid \forall \alpha. \tau$$

How are the syntax and semantics of terms extended? There are several variants, depending on whether one adopts an implicitly-typed or explicitly-typed presentation of terms and a type-passing or a type-erasing semantics.

In the explicitly-typed variant (Reynolds, 1974), there are term-level constructs for introducing and eliminating the universal quantifier (we recall the previous rules of simply-typed \(\lambda\)-calculus in gray):

$$M ::= x \mid \lambda x: \tau. M \mid M M \mid \Lambda \alpha. M \mid M \tau$$

We write \(F\) for the set of explicitly-typed terms.

Type variables are explicitly bound and appear in type environments:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha$$

![Figure 4.1: Typing rules for System F.](image-url)
We extend our previous convention to form environments: $\Gamma, \alpha$ extends $\Gamma$ with a new variable $\alpha$, provided $\alpha \not\in \text{dom}(\Gamma)$, i.e. $\alpha$ is neither in the domain nor in the image of $\Gamma$. We also require that environments be closed with respect to type variables. That is, we require $\text{ftv}(T) \subseteq \text{dom}(\Gamma)$ to form $\Gamma, x : \tau$. This additional requirement is a matter of convenience. It allows fewer judgments, since judgments with open contexts are not allowed. However, open contexts can always be closed by adding a prefix composed of a sequence of its free type variables. Hence, a loose definition of contexts (without this requirement) can also be used, and the differences would be insignificant.

Well-formedness of environments and types may be defined (recursively) by inference rules (Rule $\text{WFEnvVar}$ depends on well-formedness of types while Rule $\text{WfTypeVar}$ depends on well-formedness of environments):

$\text{WFEnvEmpty} \quad \text{WFEnvVar} \quad \text{WfEnvTvar} \quad \text{WfTypeVar} \quad \text{WfTypeArrow} \quad \text{WfTypeForall}$

\[ \frac{}{\vdash \emptyset} \quad \frac{\vdash \Gamma \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \quad \frac{\vdash \Gamma \quad \alpha \notin \text{dom}(\Gamma)}{\vdash \Gamma, \alpha} \]

\[ \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\vdash \Gamma \quad \alpha} \quad \frac{\vdash \Gamma \quad \alpha \vdash \tau_1 \quad \alpha \vdash \tau_2}{\vdash \Gamma \quad \tau_1 \rightarrow \tau_2} \quad \frac{\alpha \vdash \tau}{\vdash \Gamma \quad \forall \alpha. \tau} \]

There is a choice whether well-formedness of environments should be made explicit or left implicit in typing rules.

Explicit well-formedness amounts to adding well-formedness premises to every rule where the environment or some type that appears in the conclusion did not appear in any premise. Namely:

$\text{VAR} \quad \text{TAPP}$

\[ \frac{x : \tau \in \Gamma \vdash \Gamma}{\vdash \Gamma \quad x : \tau} \quad \frac{\Gamma \vdash M : \forall \alpha. \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash M \tau' : [\alpha \mapsto \tau'] \tau} \]

Explicit well-formedness is more precise and better suited for mechanized proofs. It is also recommended for (more) complicated type systems. However, it is a bit verbose and distracting for System F. The two styles are really equivalent. Formally, we choose to leave well-formedness implicit. However, for documentation purposes, we will indicate the well-formedness premises in the definition of typing rules.

### 4.2.2 Semantics

We need the following reduction for type abstraction:

\[ (\Lambda \alpha. M) \tau \rightarrow [\alpha \mapsto \tau]M \quad \text{(t)} \]

Then, there is a choice regarding whether type abstraction should stop the evaluation, or let reduction proceed.
Type-passing semantics  In most presentations of System $\mathcal{F}$, type abstraction blocks the evaluation and is defined as follows:

$$E ::= [] \mid V [] \mid [] \tau \quad V ::= \lambda x : \tau. M \mid \Lambda \alpha. M$$

This is a type-passing semantics. Indeed, $\Lambda \alpha.((\lambda y : \alpha. y)\ V)$ is a value while its type erasure is $(\lambda y. y)[V]$ is not—and can be further reduced.

The type-passing semantics is perhaps more natural in a language with a call-by-value semantics since type abstraction stops evaluation exactly as value abstraction.

However, it does not fit our view that the untyped semantics should pre-exist and that a type system is only a predicate that selects a subset of the well-behaved terms, since type abstraction alters the semantics.

In particular, it introduces a discontinuity between monomorphic and polymorphic types. Assume for example that $f$ is list flattening of type $\forall \alpha. \text{list} \ (\text{list} \ \alpha) \rightarrow \text{list} \ \alpha$ and $\circ$ is the composition function $\Lambda \alpha_1.\Lambda \alpha_0.\Lambda \alpha_2. \lambda f : \alpha_0 \rightarrow \alpha_2. \lambda g : \alpha_1 \rightarrow \alpha_0. \lambda x : \alpha_1. f \ g \ x$; then, the monomorphic function $(f \ \text{int}) \ (\circ \ \text{int} \ (\text{list} \ \text{int}) \ (\text{list} \ (\text{list} \ \text{int})))$ reduces to

$$\Lambda \alpha. (f \ \alpha) \ (\circ \ \alpha \ (\text{list} \ (\text{list} \ \alpha))) \ (f \ (\text{list} \ \alpha))$$

is irreducible. This discontinuity is disturbing especially in an implicitly-typed language such as ML, where type inference infers the most general version, which behaves less efficiently than its less general monomorphic variant.

Furthermore, since the type-passing semantics requires both values and types to exist at runtime, it can lead to a duplication of machinery. Compare type-preserving closure conversion in type-passing (Minamide et al., 1996) and in type-erasing (Morisset et al., 1999) styles.

Type-erasing semantics  To recover a type-erasing semantics (also called an untyped semantics), we need to allow evaluation under type abstraction:

$$E ::= [] \mid V [] \mid [] \tau \mid \Lambda \alpha. [] \quad V ::= \lambda x : \tau. M \mid \Lambda \alpha. V$$

Accordingly, we only need a weaker version of $\iota$-reduction:

$$(\Lambda \alpha. V) \ \tau \longrightarrow [\alpha \mapsto \tau]V$$

We now have:

$$\Lambda \alpha. (\lambda y : \alpha. y) \ V \longrightarrow \Lambda \alpha. V$$

We will show below that this defines a type-erasing semantics, indeed.

As an apparent drawback, the type-erasing semantics does not allow a typecase; however, typecase can be simulated by viewing runtime type descriptions as values (Crary et al., 2002).

On the opposite the type-erasing semantics, has several advantages: it does not alter the semantics of untyped terms; it coincides with the semantics of ML—and, more generally,
with the semantics of most programming languages. It also exhibits difficulties when adding side effects while the type-passing semantics keeps them hidden.

For all these reasons, we prefer the type-erasing semantics, which we chose in the rest of this course. Notice that we allow evaluation under a type abstraction as a consequence of choosing a type-erasing semantics—and not the converse.

The two views may be reconciled by restricting type abstraction to value-forms (which include values and variables), that is, by only allowing value-forms \( \Lambda \alpha.M \) when \( M \) is itself a value-form. Under this restriction, the type-passing and type-erasing semantics coincide. Indeed, closed type abstractions are then always type abstraction of values, and evaluation under type abstraction even if allowed may never be used. We will choose this restriction as a way to preserve type soundness when adding side effects to the language.

**Implicitly-typed v.s. explicitly-typed variants** We presented the explicitly-typed variant of System F. This is simpler for the meta-theoretical study while the implicitly typed version, and in particular its interesting ML subset, may be more convenient to use in practice. Fortunately, most meta-theoretical properties of the explicitly-typed version can then be transferred to the implicitly-typed version—so that proofs do not have to be redone in a different setting when putting theory into practice!

### 4.2.3 Extended System F with datatypes

System F is quite expressive: it enables the encoding of data structures. For instance, the Church encoding of pairs in the untyped \( \lambda \)-calculus is actually well-typed in System F:

\[
\begin{align*}
\text{Pair} & \triangleq \Lambda \alpha_1. \Lambda \alpha_2. \Lambda \lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. \lambda \lambda y : \alpha_1 \rightarrow \alpha_2 \rightarrow \beta. y \, x_1 \, x_2 \\
\text{proj}_i & \triangleq \Lambda \alpha_1. \Lambda \alpha_2. \lambda y : \forall \beta. (\alpha_1 \rightarrow \alpha_2 \rightarrow \beta) \rightarrow \beta. y \, \alpha_i (\lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. x_i) \\
[\text{Pair}] & \triangleq \lambda x_1. \lambda x_2. \lambda y. y \, x_1 \, x_2 \\
[\text{proj}_i] & \triangleq \lambda y. y \, (\lambda x_1. \lambda x_2. x_i)
\end{align*}
\]

Notice the use of first-class polymorphism in the definition of \( \text{proj}_i \). This is general in the encoding of datatypes.

Natural numbers, List, etc. can also be encoded.

Unit, Pairs, Sums, etc. can also be added to System F as primitives. We can then proceed as for simply-typed \( \lambda \)-calculus. However, we may also take advantage of the expressive type system of System F to deal with such extensions in a more elegant way: thanks to polymorphism, we need not add new typing rules for each extension. We may instead add one typing rule for constants and parametrize the definition by an initial typing environment \( \Delta \) for constants. This allows sharing the meta-theoretical developments between the different extensions.
Adding primitive pairs Let us first illustrate datatypes on an example, adding primitive pairs to System F. We will then generalize the presentation to parametrize the extension as suggested above.

We introduce a new type constructor \((\cdot \times \cdot)\) of arity 2 to classify pairs:
\[
\tau ::= \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \mid \tau \times \tau
\]
Expressions are extended with a constructor \((\cdot, \cdot)\) and two destructors \(\text{proj}_1\) and \(\text{proj}_2\) with the respective signatures:
\[
\begin{align*}
\text{Pair} & : \forall \alpha_1, \forall \alpha_2. \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2 \\
\text{proj}_i & : \forall \alpha_1, \forall \alpha_2. \alpha_1 \times \alpha_2 \rightarrow \alpha_i
\end{align*}
\]
that forms the initial typing environment \(\Delta\). We need not add any new typing rule, but instead type programs in the initial environment \(\Delta\).

This allows for the formation of partial applications of constructors and destructors. Hence, values are extended as follows:
\[
V ::= \ldots \mid \text{Pair} \mid \text{Pair} \tau \mid \text{Pair} \tau \tau \mid \text{Pair} \tau \tau V \mid \text{Pair} \tau \tau V V \\
\mid \text{proj}_1 \mid \text{proj}_i \mid \text{proj}_i \tau \tau
\]
We add the two following reduction rules:
\[
\text{proj}_i \tau_1 \tau_2 (\text{pair} \tau_1' \tau_2' V_1 V_2) \rightarrow V_i \quad (\delta_{\text{pair}})
\]
Notice that, for well-typed programs, \(\tau_i\) and \(\tau_i'\) will always be equal, but the reduction will not check this at runtime. This could be enforced by replacing \(\delta\) with the following rule:
\[
\text{proj}_i \tau_1 \tau_2 (\text{pair} \tau_1 \tau_2 V_1 V_2) \rightarrow V_i \quad (\delta'_{\text{pair}})
\]
The two semantics coincide on well-typed terms, but differ on ill-typed terms where \(\delta'_{\text{pair}}\) may block when rule \(\delta_{\text{pair}}\) would progress, ignoring type errors. Interestingly, using \(\delta'_{\text{pair}}\) simplifies the proof obligation in subject reduction but introduces a more stronger proof obligation in progress.

Notice that since pairs are defined by applying the pair constructor to two arguments, the programmer must first specify the types of the components although those could be uniquely determined from the arguments of the pair. Even though this is a bit more verbose that strictly necessary, it should not be considered as a problem in an explicitly-typed presentation, as removing redundant type annotations is the task of type reconstruction.

A general approach Adding other datatypes such as booleans, integers, strings, lists, trees, etc. and operations on them can be done similarly. However, all these extensions are quite similar. Hence, we propose a general approach for adding constants to System F, which can then be instantiated independently—or simultaneously—to each of the previous cases: provided the dynamic semantics of constraints agree with their static semantics (some requirements must be satisfied in order to instantiate the general approach), the soundness
of the extension then automatically follows.

We assume given a collection of constants, written with letter c, each of which given with a fix arity written \textit{arity} (c). Constants must actually be partitioned into constructors (written C) and destructors (written d); moreover, we disallow nullary destructors\footnote{Nullary polymorphic destructors introduce pathological cases to maintain the semantics type-erasing—for little benefit in return.}.

Expressions are extended with constant expressions.

\[ M ::= x \mid \lambda x: \tau. M \mid M M \mid \Lambda \alpha. M \mid M \tau \mid c \]

The difference between constructors and destructors lies in the fact that full application of constructors are values while full applications of destructors are not—they must be reduced. Partial applications of constants are always values. Hence, the following definition of values:

\[ V ::= x \mid \lambda x: \tau. M \mid \Lambda \alpha. V \mid C \tau_1 \ldots \tau_i V_1 \ldots V_n \mid d \tau_1 \ldots \tau_j V_1 \ldots V_k \]

where \( n \) is less or equal to the arity of C and \( k \) is strictly less than the arity of d. The semantics of constants is given by providing, for each destructor d a relation \( \delta_d \) defined by a set of \( \delta \)-rules of the form:

\[ d \tau_1 \ldots \tau_j V_1 \ldots V_k \rightarrow M \]  \hspace{1cm} (\( \delta_d \))

We assume given a collection of type constructors G, with their arity, written \textit{arity} (G). Types are extended as follows.

\[ \tau ::= \ldots \mid G \tau_1 \ldots \tau_n \]

We assume that types respect the arities of type constructors, \textit{i.e.} \( n \) is equal to \textit{arity} (G) in the expressions \( G \tau_1 \ldots \tau_n \).

The typing of constants is given by the initial typing environment \( \Delta \), which binds each constant c of arity \( n \) to a type of the form \( \forall \alpha_1 \ldots \forall \alpha_j. \tau_1 \rightarrow \ldots \tau_i \rightarrow \tau \). When c is a constructor C, we require that the top most type constructor of \( \tau \) not be an arrow, but some type constructor G. We then say that C is a G-constructor. We require that \( \Delta \) be well-formed (in the empty environment, hence closed). Constants are typed as variables, except that their types are looked up in \( \Delta \):

\[
\frac{\text{Cst}}{c : \tau \in \Delta} \vdash \Gamma \\
\frac{\Gamma \vdash c : \tau}{c : \tau} 
\]

Taking typing constraints into account, we may give a more restrictive characterization of well-typed values: in the presentation above \( i \) is at most the number of quantified variables in the type scheme of the constructor, and whenever \( n \) is non zero, \( i \) is equal to this number. And similarly for destructors. For instance, if C is a constructor (respectively, d is a destructor) of arity \( q \) and of type \( \forall \alpha_1 \ldots \alpha_p. \tau_1' \rightarrow \ldots \tau_q' \rightarrow \tau \), then values will contain:

\[ C \mid C \tau_1 \mid \ldots C \tau_1 \ldots C \tau_1 \mid C \tau_1 \ldots C \tau_1 \ldots C \tau_1 \ldots C \tau_1 \ldots C \tau_1 \ldots V_1 \ldots \]

\[ \ldots V_1 \ldots V_q \]
and
\[ c | c \tau_1 | \ldots c \tau_p | c \tau_1 \ldots \tau_p V_1 | \ldots c \tau_1 \ldots \tau_p V_1 \ldots V_{q-1} \]

Of course, we need assumptions to relate typing and reduction of constants.

**Definition 2** \( \delta \)-reduction is sound if it preserves typings and ensures progress for primitives. That is

- If \( \bar{\alpha} \vdash M_1 : \tau \) and \( M_1 \rightarrow_\delta M_2 \) then \( \bar{\alpha} \vdash M_2 : \tau \).
- If \( \bar{\alpha} \vdash M_1 : \tau \) and \( M_1 \) is of the form \( d\tau_1 \ldots \tau_k V_1 \ldots V_n \) where \( n = \text{arity} (d) \), then there exists \( M_2 \) such that \( M_1 \rightarrow_\delta M_2 \).

Intuitively, progress for constants means that the domain of destructors is at least as large as specified by their type in \( \Delta \).

We will show below that soundness of \( \delta \)-rules is sufficient to ensure soundness of the extension.

For example, to add a unit constant, we only introduce a type constant `unit` and a constructor \((\)\) of arity 0 of type `unit`. As no primitive is added, \( \delta \)-reduction is obviously sound. Hence, the extension of System \( F \) with unit is sound.

**Exercise 24 (Pairs as constants)** Reformulate the extension of System \( F \) with pairs as constants. Check soundness of the \( \delta \)-rules.

**(Solution p. 84)**

**Exercise 25 (Conditional)** Give a presentation of boolean with a conditional as constants. Is this sound? Isn’t there something wrong? Would you know how to fix it?

**(Solution p. 84)**

**Exercise 26 (List)** 1) Formulate the extension of System \( F \) with lists as constants.

**(Solution p. 84)** 2) Check that this extension is sound.

**Extending System \( F \) with a fixpoint** The call-by-value fixpoint combinator \( Z \) (see §2) is not typable in System \( F \)—indeed this would allow program to loop while all programs terminate in System \( F \).

However, we may introduce a fixpoint as a binary primitive with the following typing assumption:
\[
\text{fix} : \forall \alpha. \forall \beta. ( (\alpha \to \beta) \to \alpha \to \beta ) \to \alpha \to \beta \quad \in \Delta
\]

and the reduction rule:
\[
\text{fix } \tau_1 \tau_2 V_1 V_2 \rightarrow V_1 (\text{fix } \tau_1 \tau_2 V_1) V_2 \quad (\delta_{\text{fix}})
\]

It is straightforward to check the soundness of this extension: Progress is by construction, since \( \text{fix} \) does not destruct values. As for subject reduction, assume \( \Gamma \vdash \text{fix } \tau_1 \tau_2 V_1 V_2 : \tau \).

By inversion of typing rules, \( \tau \) must be equal to \( \tau_2 \). \( V_1 \) and \( V_2 \) must be of respective types \( (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \) and \( \tau_1 \) in the typing context \( \Gamma \). We may then easily build a derivation of the judgment \( \Gamma \vdash V_1 (\text{fix } \tau_1 \tau_2 V_1) V_2 : \tau \).
Exercise 27 In ML a one-constructor datatype can be used to emulate recursive types, namely a type Any such that a value of type \( \text{any} \to \text{any} \) can be converted to a value of type \( \text{any} \), and conversely. Give the definition in ML. Describe the extension as the addition of new constants. Verify the soundness of \( \delta \)-rules.

Use this extension to define a call-by-value fixpoint operator of type

\[
((\text{any} \to \text{any}) \to \text{any} \to \text{any}) \to \text{any} \to \text{any}
\]

in ML without using \texttt{let rec} or implicit recursive types (the \texttt{--rectypes} option). (See Exercise 7 for a definition of the fix-point in the \( \lambda \)-calculus or in ML with recursive types.)

(Solution p. 85)

4.3 Type soundness

We proof type soundness for System F with constants, assuming the soundness of \( \delta \)-reduction.

The structure of the proof is similar to the case of simply-typed \( \lambda \)-calculus and follows from subject reduction and progress. Subject reduction uses the following auxiliary lemmas: inversion of typing rules (Lemma 17), permutation (Lemma 18), weakening (Lemma 19), expression substitution (Lemma 20), type substitution (Lemma 21), and compositionality of typing (Lemma 22).

Lemma 17 (Inversion of typing rules) Assume \( \Gamma \vdash M : \tau \).

- If \( M \) is a variable \( x \), then \( x \in \text{dom}(\Gamma) \) and \( \Gamma(x) = \tau \).
- If \( M \) is \( \lambda x : \tau_0 . M_1 \), then \( \tau \) is of the form \( \tau_0 \to \tau_1 \) and \( \Gamma, x : \tau_0 \vdash M_1 : \tau_1 \).
- If \( M \) is \( M_1 M_2 \) then \( \Gamma \vdash M_1 : \tau_2 \to \tau \) and \( \Gamma \vdash M_2 : \tau_2 \) for some type \( \tau_2 \).
- If \( M \) is a constant \( c \), then \( c \in \text{dom}(\Delta) \) and \( \Delta(x) = \tau \).
- If \( M \) is \( M_1 \tau_2 \) then \( \tau \) is of the form \( [\alpha \mapsto \tau_2] \tau_1 \) and \( \Gamma \vdash M_1 : \forall \alpha . \tau_1 \).
- If \( M \) is \( \Lambda \alpha . M_1 \), then \( \tau \) is of the form \( \forall \alpha . \tau_1 \) and \( \Gamma, \alpha \vdash M_1 : \tau_1 \).

Lemma 18 (Permutation) If \( \Gamma \) and \( \Gamma' \) are two well-formed permutations, then \( \Gamma \vdash M : \tau \) iff \( \Gamma' \vdash M : \tau \).

Proof: Formally, the proof is by induction on \( M \). The key is the observation that when \( \Gamma \) and \( \Gamma' \) are both well-formed and permutations of one another, they are equivalent as partial functions, i.e. they give the same bindings and can be extended in the same manner.

Lemma 19 (Weakening) If \( \Gamma \vdash M : \tau \) and \( \Gamma \vdash \Gamma, \Gamma' \), then \( \Gamma, \Gamma' \vdash M : \tau \).
4.3. TYPE SOUNDNESS

Proof: It suffices to prove the lemma when $\Gamma'$ is either $x : \tau'$ or $\alpha$, since the general case follows by induction on the length of $\Gamma'$. We may prove both simultaneously, by induction on $M$. The proof is similar to the one for simply-typed $\lambda$-calculus—we just have more cases. Cases for value and type abstraction appeal to the permutation lemma. More precisely:

Case $M$ is $y$: By inversion of typing, the judgment must be derived with rule $\text{Var}$ hence $y : \tau$ is in $\Gamma$ and a fortiori $y : \tau$ is in $\Gamma, \Gamma'$. We may thus conclude by rule $\text{Var}$.

Case $M$ is $c$: By inversion of typing, the judgment must be derived with rule $\text{Cst}$ hence we have $y : \tau$ is in $\Delta$ and we may conclude with rule $\text{Cst}$.

Case $M$ is $\lambda y : \tau_1.M_2$: W.l.o.g. we may choose $y$ disjoint from $\Gamma$ and $\Gamma'$ (1). By inversion of typing, the judgment must be derived with rule $\text{Abs}$ hence $\Gamma, y : \tau_1 \vdash M_1 : \tau_2$ where $\tau$ is $\tau_1 \rightarrow \tau_2$. Since $\Gamma, y : \tau$ is well-formed, by (1), both $\Gamma, y : \tau_1, \Gamma'$ and $\Gamma, \Gamma', y : \tau_1$ are well-formed (2). By induction hypothesis, we have $\Gamma, \Gamma', x : \tau_1 \vdash M_1 : \tau_2$. Using the permutation lemma and (2), we have $\Gamma, \Gamma', x : \tau_1 \vdash M_1 : \tau_2$. We conclude with rule $\text{Abs}$.

Case $M$ is $\Lambda \beta.M_1$: W.l.o.g. we may choose $\beta$ disjoint from $\Gamma$ and $\Gamma'$ (3). By inversion of typing, the judgment must be derived with rule $\text{Tabs}$ hence $\Gamma, \beta \vdash M_1 : \tau_1$ with $\forall \beta.\tau_1$ equal to $\tau$. Since $\Gamma, \beta$ is well-formed, by (3), both $\Gamma, \beta, \Gamma'$ and $\Gamma, \Gamma', \beta$ are well-formed (4). By induction hypothesis, we have $\Gamma, \beta, \Gamma' \vdash M_1 : \tau_1$. We use the permutation lemma to obtain $\Gamma, \Gamma', \alpha \vdash M_1 : \tau_1$ and conclude with Rule $\text{Tabs}$.

Case $M$ is $M_1\ M_2$ or $M_1\ \tau_1$: By inversion of typing, induction hypothesis applied to the premises, and $\text{App}$ or $\text{TApp}$ to conclude.

Lemma 20 (Expression substitution, strengthened)

If $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ and $\Gamma \vdash M_0 : \tau_0$ then $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$.

We have strengthened the lemma with an arbitrary context $\Gamma'$ as for the simply-typed $\lambda$-calculus. We have also generalized the lemma with an arbitrary context $\Gamma$ on the left and an arbitrary expression $M$, as this does not complicate the proof (and the stronger result will be used later). The proof is similar to the one for the simply-typed $\lambda$-calculus, with just a few more cases.

Exercise 28 Write the details of the proof.

Lemma 21 (Type substitution, strengthened)

If $\Gamma, \alpha, \Gamma' \vdash M : \tau$ and $\Gamma \vdash \tau_0$ then $\Gamma, \theta \Gamma' \vdash \theta M : \theta \tau$ where $\theta$ is $[\alpha \mapsto \tau_0]$.

As for expression substitution, we have strengthened the lemma and generalized it using an arbitrary environment instead of the empty environment, as it does not complicate the proof, but yields a stronger result. This lemma resembles the one for expression substitutions.
However, the substitution must also apply to the environment $\Gamma'$ and the result type $\tau$ since $\alpha$ may appear free in them.

The proof is by induction on $M$. The interesting cases are for type and value abstraction, which required the strengthened version with an arbitrary typing context $\Gamma'$ on the right. Then, the proof is straightforward. \(\text{(Details of the proof p. 86)}\)

**Exercise 29** Write the details of the proof.

**Lemma 22 (Compositionality)** If $\Gamma \vdash E[M] : \tau$, then there exists $\bar{\alpha}$ and $\tau'$ such that $\Gamma, \bar{\alpha} \vdash M : \tau'$ and all $M'$ verifying $\Gamma, \bar{\alpha} \vdash M' : \tau'$ also verify $\Gamma \vdash E[M'] : \tau$.

**Proof**: The proof is by case on $E$. Each case is easy. The main difference with the simply-typed $\lambda$-calculus is that the case for type abstraction $\Lambda \alpha.E_0$ requires to extend the environment with type variables.

**Theorem 9 (Subject Reduction)** Reduction preserves typings. If $\Gamma \vdash M : \tau$ and $M \rightarrow M'$ then $\Gamma \vdash M' : \tau$.

The proof is by induction over the derivation of $M \rightarrow M'$. Using the previous lemmas and the subject-reduction assumption for $\delta$-reduction, the proof is straightforward.

**Proof**: By induction over the derivation of $M \rightarrow M'$, then by inversion of the typing derivation of $\Gamma \vdash M : \tau$ (1).

Case $(\lambda x : \tau_1. M_1) V \rightarrow [x \mapsto V]M_1$: By inversion, the typing derivation of (1) is of form:

\[
\begin{align*}
\Gamma, x : \tau' &\vdash M_1 : \tau' & (2) \\
\Gamma &\vdash (\lambda x : \tau'. M_1) : \tau' \rightarrow \tau & (3) \\
\Gamma &\vdash V : \tau' \\
\Gamma \vdash (\lambda x : \tau'. M_1) V : \tau & (1)
\end{align*}
\]

The value-substitution Lemma applied to (2) and (3) gives the expected result.

Case $(\Lambda \alpha.V) \tau_0 \rightarrow [\alpha \mapsto \tau_0]V$: By inversion of (1), we have $\Gamma, \alpha \vdash V : \tau_1$ (4) where $\tau$ is $[\alpha \mapsto \tau_0]\tau_1$. The type-substitution Lemma applied to (4) gives the expected result $\Gamma \vdash [\alpha \mapsto \tau_0]V : \tau$.

Case $E[M_0] \rightarrow E[M'_0]$: The hypothesis is $M_0 \rightarrow M'_0$. Assume $\Gamma \vdash E[M_0] : \tau$. By compositionality, there is some type $\tau_0$ and type variables $\bar{\alpha}$ such that $\Gamma, \bar{\alpha} \vdash M_0 : \tau_0$ (5) and for all $M'_0$ such that $\Gamma, \bar{\alpha} \vdash M'_0 : \tau_0$, we have $\Gamma \vdash E[M'_0] : \tau$. Therefore it suffices to show $\Gamma, \bar{\alpha} \vdash M'_0 : \tau_0$, which holds by induction hypothesis applied to (5).
The classification lemma, which is a key to progress, is slightly modified to account for polymorphic types and constructed types.

**Lemma 23 (Classification)** Assume $\bar{\alpha} \vdash V : \tau$

- If $\tau$ is an arrow type, then $V$ is either a function or a partial application of a constant to values.
- If $\tau$ is a polymorphic type, then $V$ is either a type abstraction of a value or a partial application of a constant to types.
- If $\tau$ is a constructed type, then $V$ is constructed value.

The last case can be refined by partitioning constructors into their associated type-constructor: If the top-most type constructor of $\tau$ is $G$, then $V$ is a value constructed with a $G$-constructor.

The proof is similar to the one for simply-typed $\lambda$-calculus.

Progress is restated as follows:

**Theorem 10 (Progress, strengthened)** A well-typed, irreducible closed term is a value: if $\bar{\alpha} \vdash M : \tau$ and $M \not\rightarrow$, then $M$ is some value $V$.

The theorem has been strengthened, using a sequence of type variables $\bar{\alpha}$ for the typing context instead of the empty environment. It can then be proved by induction and case analysis on $M$, relying mainly on the classification lemma and the progress assumption for $\delta$-reduction.

**Proof:** By induction on (the derivation of) $M$. Assume $\bar{\alpha} \vdash M : \tau$ and $M$ is irreducible.

*Case $M$ is $x$: This is not possible since $x$ is not well-typed in $\bar{\alpha}$.*

*Case $M$ is $c$: Then $M$ is a value (a fully applied constructor or a partially applied destructor), as expected.*

*Case $M$ is $\lambda x : \tau. M_1$: Then $M$ is a value, as expected.*

*Case $M$ is $M_1 M_2$: Then, $\bar{\alpha} \vdash M_1 : \tau_2 \rightarrow \tau_1$, and $\bar{\alpha} \vdash M_2 : \tau_2$. Since the left application is an evaluation context, $M_1$ is irreducible. Hence, by induction hypothesis, $M_1$ is a value. Since the right application of a value is an evaluation context, $M_2$ is also a value. Hence, by induction hypothesis, $M_2$ is irreducible. Since it has an arrow type, it follows from the classification lemma that it is a partial application of a constant to values. Hence, $M$ is itself the application of a constant to values. Since it cannot be reduced, it follows from the progress assumption for $\delta$-rules that it is not a full application of a destructor. Hence, it is either a full application of a constructor or a partial application of a constant to values. In both cases, $M$ is a value.*

*Case $M$ is $\Lambda \beta. M_1$: Then, $\bar{\alpha}, \beta \vdash M_1 : \tau_1$. Since type abstraction is an evaluation context $M_1$ is irreducible. Hence, by induction hypothesis, $M_1$ is a value and so is $M$.*
Case $M$ is $M_1 \tau_1$: Then, $\bar{\alpha} \vdash M_1 : \forall \alpha. \tau_2$ with $\tau$ equal to $[\alpha \mapsto \tau_1] \tau_2$. Since type application is an evaluation context, $M_1$ is irreducible. Hence, by induction hypothesis, $M_1$ is a value. Since $M$ is irreducible $M_1$ is not a type abstraction. Since $M_1$ has a polymorphic type, it follows from the classification lemma that $M_1$ is an application of a constant $c$ to types (as it is not a type abstraction). Since it is irreducible, it follows from the progress assumption for $\delta$-rules that $c$ is a destructor or the application is partial. In both cases $M$ is a value.

**Theorem 11 (Normalization)** Reduction terminates in pure System F.

This is also true for arbitrary reductions and not just for call-by-value reduction. This is a difficult proof, which generalizes the proof method for the simply-typed $\lambda$-calculus. It is due to Girard (1972) (see also Girard et al. (1990)).

### 4.4 Type erasing semantics

We have presented the explicitly-typed variant of System F. In this section, we verify that this semantics is type erasing. Hence, there is an implicitly-typed presentation of System F.

#### 4.4.1 Implicitly-typed System F

The implicitly-typed version of System F, can be defined as follows. The syntax of terms and their dynamic semantics are those of the untyped $\lambda$-calculus extended with constants. However, we only accept a subset of terms of the $\lambda$-calculus, retaining only those that are the type erasure of a term in F.

We write $\llbracket F \rrbracket$ for the set of implicitly-typed terms and $F$ for the set of explicitly-typed terms. We use letters $a$, $v$, and $e$ to range over implicitly-typed terms, values, and evaluation contexts, reusing the same notations as for the untyped $\lambda$-calculus.

The set of terms may also be characterized by typing rules that operate directly on unannotated terms. These are obtained from the typing rules of F by dropping all type information in terms. They are presented in Figure 4.2. We use the prefix $\text{IP}$ to distinguish them from the typing rules for explicit System F.

Unsurprisingly, as a result of erasing type information in terms, the rules that introduce and eliminate the universal quantifier are no longer syntax-directed.

**Remark 4** Notice that the explicit introduction of variable $\alpha$ in the premise of Rule $\text{Tabs}$ contains an implicit side condition $\alpha \notin \Gamma$ due to the assumption on the formation of typing environments.

In implicitly-typed System F, as in ML, the introduction of type variables in typing context is often left implicit. (In some extensions of System F, type variables may carry a
4.4. TYPE ERASING SEMANTICS

kind or a bound and must be explicitly introduced.) If we chose to do so, we would need an explicit side-condition on Rule \textbf{Tabs} as follows:

\[ \Gamma \vdash a : \tau \quad \alpha \neq \Gamma \]

Omitting the side condition would lead to unsoundness. Below on the left-hand side is a type derivation for a type cast (\textit{Obj.magic} in \textit{OCaml}), which is equivalent to using an ill-formed context (on the right-hand side):

A good intuition is that a judgment \( \Gamma \vdash a : \tau \) corresponds to the logical assertion \( \forall \bar{\alpha}. (\Gamma \Rightarrow (a : \tau)) \), where \( \bar{\alpha} \) are the free type variables of the judgment. In this view, \textbf{Tabs-Bis} corresponds to the axiom:

\[ \forall \bar{\alpha}. (P \Rightarrow Q) \equiv P \Rightarrow (\forall \bar{\alpha}. Q) \]

which without the side condition is obviously wrong.

The next lemma, states that the two definitions of \([F]\)—or, equivalently, the two type systems for implicitly-typed System \( F \) and explicitly type System \( F \)—coincide. The proof is immediate.

\textbf{Lemma 24} \( \Gamma \vdash a : \tau \) in implicitly-typed System \( F \) if and only if there exists an explicitly-typed expression \( M \) whose erasure is a such that \( \Gamma \vdash M : \tau \).
For example, consider the term \( a_0 \) in \( \mathbb{F} \) equal to \( \lambda f x y . (f \ x, f \ y) \). A version that carries explicit type abstractions and annotations is:

\[
\Lambda \alpha_1 . \Lambda \alpha_2 . \lambda f : \alpha_1 \to \alpha_2 . \lambda x : \alpha_1 . \lambda y : \alpha_1 . (f \ x, f \ y)
\]

Unsurprisingly, this term admits the polymorphic type:

\[
\tau_1 \triangleq \forall \alpha_1 . \forall \alpha_2 . (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2
\]

Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

\[
\Lambda \alpha_1 . \Lambda \alpha_2 . \lambda f : \forall \alpha . \alpha \to \alpha . \lambda x : \alpha_1 . \lambda y : \alpha_2 . (f \ \alpha_1 \ x, f \ \alpha_2 \ y)
\]

This term admits the polymorphic type:

\[
\tau_2 \triangleq \forall \alpha_1 . \forall \alpha_2 . (\forall \alpha . \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2
\]

This begs the question: which of the two types \( \tau_1 \) or \( \tau_2 \) is more general? Type \( \tau_1 \) requires the second and third arguments to admit a common type, while type \( \tau_2 \) requires the first argument to be polymorphic.

**Exercise 30** Find two terms \( a_1 \) and \( a_2 \) such that \( a_1 \) has type \( \tau_1 \) type but not type \( \tau_2 \), and conversely for \( a_2 \). (Just give the terms \( a_1 \) and \( a_2 \), you do not have to prove well-typedness or ill-typedness.

(Solution p. 86)

This suggests that the two types are not comparable, that is, neither one can be an instance of the other.

Intuitively, one may think semantically of (i.e. interpret) a closed type as the set of terms of that type, and of instance as inclusion between types. With such a view in mind then \( \tau_1 \) and \( \tau_2 \) are indeed incomparable. This does not imply that \( a_0 \) does not have a principal type: there could exist a type \( \tau_0 \) that contains \( a_0 \) and that is included in the intersection of (the interpretations of) \( \tau_1 \) and \( \tau_2 \). Indeed, one can do so in a richer system, such as System F\( ^{\omega} \).

**Exercise 31** In System F\( ^{\omega} \), find a type \( \tau_0 \) for \( a_0 \) that is more general than both \( \tau_1 \) and \( \tau_2 \).

(Solution p. 86)

### 4.4.2 Type instance

To reason formally, we must first define what it means for \( \tau_2 \) to be an instance of \( \tau_1 \)—or, equivalently, for \( \tau_1 \) to be more general than \( \tau_2 \). Several definitions are possible. In System F, to be an instance is usually defined by the rule:

\[
\text{INST-GEN} \quad \\frac{\beta \not\equiv \forall \vec{\alpha} . \tau}{\forall \vec{\alpha} . \tau \leq \forall \beta . [\vec{\alpha} \mapsto \tau] \tau}
\]
One can show that, if $\tau_1 \leq \tau_2$, then any term that has type $\tau_1$ has also type $\tau_2$; that is, the following rule is admissible in the implicitly-typed version:

$$
\text{Sub} \quad \frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}
$$

Perhaps surprisingly, the rule is not derivable in our presentation of System F. Although, we have the following derivation,

$$
\text{Inst}^* \quad \frac{\Gamma, \tilde{\beta} \vdash a : \forall \alpha. \tau}{\text{Gen}^* \quad \frac{\Gamma, \beta \vdash a : [\alpha \mapsto \tilde{\tau}] \tau}{\Gamma \vdash a : \forall \beta.[\alpha \mapsto \tilde{\tau}] \tau}}
$$

the premise $\Gamma, \tilde{\beta} \vdash a : \forall \alpha. \tau$ can only be justified from the assumption $\Gamma \vdash a : \forall \alpha. \tau$ by an application of weakening (the side condition $\tilde{\beta} \not\# \forall \alpha. \tau$ of rule Gen ensures that $\Gamma, \tilde{\beta}$ is well-formed.) Otherwise, in context $\Gamma$ alone, $\tilde{\tau}$ would not necessarily be well-formed, as required by rule Gen.

However, in a version of System F that does not introduce type variables explicitly in $\Gamma$, then weakening of type variables would be built-in and implicit and the rule Sub would become derivable. (This shows that the notion of derivability is somewhat fragile as it depends on the presentation of the rules.)

We may also wonder what is the counterpart of the instance relation in explicitly-typed System F. Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \leq \tau_2$. How can we see $M$ with type $\tau_2$? Since explicitly-typed terms have unique types, the term $M$ of type $\tau_1$ cannot itself also have type $\tau_2$. However, we can wrap $M$ with a retyping context that transforms a term of type $\tau_1$ to one of type $\tau_2$. Since $\tau_1 \leq \tau_2$, the types $\tau_1$ and $\tau_2$ must be of the form $\forall \alpha. \tau$ and $\forall \beta.[\alpha \mapsto \tilde{\tau}] \tau$ where $\tilde{\beta} \not\# \forall \alpha. \tau$. W.l.o.g, we may assume that $\tilde{\beta} \not\# \Gamma$ (6), as it may always be satisfied up to a renaming of bound variables $\beta$. Then, we have the pseudo-derivation on the left-hand side (where the weakening lemma is used as a pseudo-typing rule Weakening), which can be abbreviated by the admissible typing rule Sub given on the right-hand side.

$$
\text{Admissible rule:} \quad \frac{\Gamma \vdash M : \forall \alpha. \tau \quad \tilde{\beta} \not\# \forall \alpha. \tau}{\text{Sub} \quad \frac{\Gamma \vdash M : \forall \alpha. \tau \quad \beta \not\# \Gamma}{\Gamma \vdash \Lambda \beta. M : \forall \beta.[\alpha \mapsto \tilde{\tau}] \tau}}
$$

In F, we rather write subtyping as a judgment $\Gamma \vdash \tau_1 \leq \tau_2$ instead of the binary relation $\tau_1 \leq \tau_2$.

---

2 A rule is admissible if adding the rule does not change the validity of judgments. That is, it may just allow for more derivations of already valid judgments.

3 A rule is derivable if it can be replaced by a sub-derivation tree with the same premises and conclusion.
to also mean \( \Gamma \vdash \tau_1 \) and \( \Gamma \vdash \tau_2 \) and so simultaneously keep track of the well-formedness of types.

In the previous example, the subtyping judgment \( \Gamma \vdash \tau_1 \leq \tau_2 \) has been witnessed by the wrapping context \( \Lambda \vec{\beta}.[\cdot] \vec{\tau} \). Since this context is only composed of type abstractions and type applications, it changes the type of the term put in the hole without changing its behavior and it is called a \textit{retyping context}. More generally, we may allow arbitrary wrappings of type abstractions and type applications around expressions. As in the example, they never change the type erasure. Retyping contexts are thus defined by the following grammar:

\[
R ::= [] / \Lambda \alpha. R / R \tau
\]

(Notice that retyping contexts are arbitrarily deep here, by contrast with single-node evaluation contexts \( E \) defined earlier.)

We could also define a typing judgment \( \Gamma \vdash R[\tau_1] : \tau_2 \) for retyping contexts as equivalent to \( \Gamma, x : \tau_1 \vdash R[x] : \tau_2 \) whenever \( x \) does not appear in \( R \)—or using primitive typing rules. Then, the following property holds by compositionality of typing: if \( \Gamma \vdash M : \tau_1 \) and \( \Gamma \vdash R[\tau_1] : \tau_2 \), then \( \Gamma \vdash R[M] : \tau_2 \).

We can now give another equivalent definition of subtyping, based on retyping contexts:

\( \Gamma \vdash \tau_1 \leq \tau_2 \) if and only if there exists a retyping context \( R \) such that \( \Gamma \vdash R[\tau_1] : \tau_2 \).

Notice that retyping contexts (e.g. type-instance) can only change topmost polymorphism. In particular, they cannot weaken the result types of functions or strengthen the types of their arguments.

### 4.4.3 Type containment in System \( \mathbf{F}_\eta \)

Type containment is another, more expressive, syntactic notion of instance, introduced by Mitchell (1988), that can also transform inner parts of types. It can be defined syntactically by the following set of rules:

\[
\text{Inst-Gen} \quad \begin{array}{c}
\beta \neq \forall \alpha. \tau \\
\forall \alpha. \tau \leq \forall \beta.[\alpha \mapsto \vec{\tau}]\tau
\end{array}
\quad \text{Distributivity} \quad \begin{array}{c}
\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2) \\
\forall \alpha. \tau_1 \leq \forall \alpha. \tau_2
\end{array}
\quad \text{Congruence-\(\rightarrow\)} \quad \begin{array}{c}
\tau_2 \leq \tau_1 \quad \tau_1 \leq \tau_2' \\
\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'
\end{array}
\quad \text{Congruence-\(\forall\)} \quad \begin{array}{c}
\tau_1 \leq \tau_2 \\
\forall \alpha. \tau_1 \leq \forall \alpha. \tau_2
\end{array}
\quad \text{Transitivity} \quad \begin{array}{c}
\tau_2 \leq \tau_3 \\
\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3
\end{array}
\]

With this larger instance relation, Rule [Sub] is no longer admissible—as it allows to type more terms. However, it remains sound. That is, adding Rule Sub as a primitive typing rule does not break type soundness. The resulting type system is known as System \( \mathbf{F}_\eta \), since it is also the closure of System \( \mathbf{F} \) by \( \eta \)-expansion; that is, a term is in System \( \mathbf{F}_\eta \) if and only if it is the \( \eta \)-conversion of a term in System \( \mathbf{F} \).

**Exercise 32** 1) Show that \( \forall \alpha. \tau \equiv \tau \). 2) Show that rule [Distributivity] can be replaced by
the weaker rule:

\[
\frac{\alpha \notin \text{ftv}(\tau_1)}{\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq \tau_1 \rightarrow (\forall \alpha. \tau_2)}
\]

(Solution p. 67)

One may wonder what System $F_\eta$ brings to System $F$ that it does not already have. Consider the identity function $id$ in $[F]$; it has type $\forall \alpha. \alpha \rightarrow \alpha$ but also many other incomparable types. For example, it has type $(\forall \alpha. \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$—even though a function of that type can never be applied, as there is no value of type $\forall \alpha. \alpha$ that could be passed as argument; it also has the more interesting type $\forall \alpha. (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. While these types are incomparable in $[F]$, they become comparable in System $F_\eta$. For example, in System $F_\eta$, we have:

\[
\tau_{id} \leq \left( (\forall \alpha. \alpha) \rightarrow (\forall \alpha. \alpha) \leq (\forall \alpha. \alpha) \rightarrow \tau_{id} \right) \leq \forall \beta. (\forall \alpha. \alpha) \rightarrow (\beta \rightarrow \beta)
\]

The type $\forall \alpha. \alpha \rightarrow \alpha$ is actually a principal type for $id$ in System $F_\eta$. Similarly, the function $ch$ defined below has a principal type in System $F_\eta$:

\[
ch \triangleq \lambda x. \lambda y. \text{if } M \text{ then } x \text{ else } y : \forall \beta. \beta \rightarrow \beta \rightarrow \beta
\]

Still, many expressions do not have most general types in System $F_\eta$. To see the difficulty, consider the application $\text{chid}$ of $ch$ to $id$. How can it be typed? If we keep $id$ polymorphic, then $\text{chid}$ has type $(\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$, say $\tau_1$; if, on the opposite, we instantiate $id$, then $\text{chid}$ has type $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$, say $\tau_2$—as in ML where type schemes are automatically instantiated when used. These two types are incomparable in System $F$. Although, we have $\tau_1 \leq \tau_2$ in System $F_\eta$ (as witnessed by the coercion context $\lambda x:\forall \alpha. \alpha \rightarrow \alpha. \lambda \alpha. ([\tau_2] \alpha) (x \alpha)$) and can thus give $\text{chid}$ the type $\tau_2$ and still used it at type $\tau_1$, this is more by chance than the general case: If we replace $ch$ by $ch_3$, which chooses between three arguments, then $ch_3 id$ does not have a principal type in System $F_\eta$.

System $F_\eta$ increases the expressiveness of System $F$ by enriching its type instance relation—without modifying the language of types (and other typing rules than $\text{SUB}$).

To obtain even more principal types, Le Botlan and Rémy (2009) have suggested that the language of types should be enriched with a new form of quantification $\forall \alpha \geq \tau_1. \tau_2$ to mean, intuitively, the set of types $[\alpha \rightarrow \tau] \tau_2$ when $\tau$ ranges over the set of instances of $\tau_1$. This internalizes the instance relation within the language of types. This allows to give $chid$ the type $\forall (\beta \geq \forall \alpha. \alpha \rightarrow \alpha). \beta \rightarrow \beta$ and recovering $(\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$ and $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ by choosing particular instances of $\forall \alpha. \alpha \rightarrow \alpha$ for $\beta$. By contrast with System $F_\eta$, this approach also works for the more general example of $ch_3 id$.

The language MLF has been designed for partial type reconstruction where programs are partially annotated. The user need only to provide the types of parameters of functions that are used polymorphically. The type systems is setup to implicitly use available polymorphism but never guess polymorphism. Available polymorphism comes either from type
generalization as in ML or from user-provided type annotations. Every expression has a principal type—according to the given type annotations. See (Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008) for details.

4.4.4 A definition of principal typings

A typing of an expression $M$ is a pair $\Gamma, \tau$ such that $\Gamma \vdash M : \tau$. Ideally, a type system should satisfy the principal typings property (Wells, 2002):

Every well-typed term $M$ admits a principal typing— one whose instances are exactly the typings of $M$.

Whether this property holds depends on a definition of instance. The more liberal the instance relation, the more hope there is of having principal typings.

The instance relations we have previously considered are defined syntactically. The absence of principal typings with respect to a syntactic definition of instance may result from a bad choice of the instance relation. To avoid arbitrariness, Wells (2002) introduced a more semantic notion of instance. He notes that, once a type system is fixed, a most liberal notion of instance can be defined, a posteriori, by:

A typing $\theta_1$ is more general than a typing $\theta_2$ if and only if every term that admits $\theta_1$ admits $\theta_2$ as well.

This is the largest reasonable notion of instance: $\leq$ is defined as the largest relation such that a subtyping principle is admissible.

This definition can be used to prove that a system does not have principal typings, under any reasonable definition of “instance”. Then, which systems have principal typings? The simply-typed $\lambda$-calculus has principal typings, with respect to a substitution-based notion of instance (See lesson on type inference). Wells (2002) shows that neither System $F$ nor System $F_{\eta}$ have principal typings. It was shown earlier that System $F_{\eta}$’s instance relation is undecidable (Wells, 1995; Tiuryn and Urzyczyn, 2002) and that type inference for both System $F$ and System $F_{\eta}$ is undecidable (Wells, 1999).

There are still a few positive results. Some systems of intersection types have principal typings (Wells, 2002) — but they are very complex and have yet to see a practical application. A weaker property is to have principal types. Given an environment $\Gamma$ and an expression $M$ is there a type $\tau$ for $M$ in $\Gamma$ such that all other types of $M$ in $\Gamma$ are instances of $\tau$. Damas and Milner’s type system (coming up next) does not have principal typings but it has principal types and decidable type inference.

4.4.5 Type soundness for implicitly-typed System $F$

Subject reduction and progress imply the soundness of the explicitly-typed version of System $F$. What about the implicitly-typed version? Can we reuse the soundness proof for the explicitly-
4.4. TYPE ERASING SEMANTICS

typed version? Can we pullback subject reduction and progress from F to [F]?

For progress, given a well-typed term a in [F], can we find a term M in F whose erasure is a and such that M is a value or reduces, and so conclude that a is a value or reduces? For subject reduction, given a term a₁ of type τ in [F] that reduces to a₂, can we find a term M₁ in F whose erasure is a₁ and show that M₁ reduces to a₂ to conclude that the type of a₂ is the type of a₁? In both cases, this reasoning requires a type-erasing semantics. We claimed that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped by type erasure, hence the two reductions cannot coincide exactly. The way to formalize this is to split reduction steps into βδ-steps corresponding to β or δ rules that must be preserved by type erasure, and ι-steps corresponding to the reduction of type applications that disappear during type erasure. This can be summarized in the following diagram:

We say that we establish a bisimulation between reduction on typed-terms and their erasure up to ι-steps. The bisimulation can be decomposed into a forward and a backward simulation.

Lemma 25 (Forward simulation) The reduction in F is simulated in [F] up to ι-steps. Assume Γ ⊢ M : τ. Then:
1) If M →ᵣ M', then [M] = [M']
2) If M →βδ M', then [M] →βδ [M']

The backward direction is more delicate to state, since type erasure is not bijective: there are usually many expressions of F whose type erasure is a given expression in [F].

Lemma 26 (Backward simulation) Assume Γ ⊢ M : τ and [M] → a. Then, there exists a term M' such that M →⁺ᵣ→βδ M' and [M'] = a.

Of course, the semantics can only be type erasing if δ-rules do not themselves depend on type information. First, we need δ-reduction to be defined on type erasures. We may prove the theorem directly for some concrete examples of δ-reduction.

However, keeping δ-reduction abstract is preferable to avoid repeating the same reasoning many times. Then, we must assume that it is such that type erasure establishes a bisimulation for δ-reduction taken alone.
**Assumption on $\delta$.** We assume that for any explicitly-typed term $M$ of the form $d\tau_1 \ldots \tau_j V_1 \ldots V_k$ such that $\Gamma \vdash M : \tau$, both of the following properties hold:

- **(Forward bisimulation)** If $M \rightarrow_\delta M'$, then $[M] \rightarrow_\delta [M']$.
- **(Backward bisimulation)** If $[M] \rightarrow_\delta a$, then there exists $M'$ such that $M \rightarrow_\delta M'$ and $a$ is the type-erasure of $M'$.

In most cases, the assumption on $\delta$-reduction is obvious to check. Notice however, that in general the $\delta$-reduction on untyped terms is larger than the projection of $\delta$-reduction on typed terms, because it pattern matches on the shapes of values but ignoring types. However, if we restrict $\delta$-reduction to implicitly-typed terms, then it usually coincides with the projection of reduction of explicitly-typed terms.

**Exercise 33** Consider the explicitly-typed System $F$ with pairs of the exercise 24 (p. 57). Add pairs in the untyped $\lambda$-calculus. Show that $\delta$-reduction in the untyped $\lambda$-calculus is larger than the image of the $\delta$-reduction in the explicitly-typed calculus. Verify that type erasure is a bisimulation for $\delta$-reduction.

The forward simulation (Lemma 25) is straightforward to establish. The backward simulation is slightly more delicate because there may be many antecedents of a given type erasure. We use a few easy helper lemmas to keep the proof clearer.

**Lemma 27**

1) A term that erases to $\bar{e}[a]$, then $M_0$ is of the form $\bar{E}[M]$ where $[\bar{E}]$ is $\bar{e}$ and $[M]$ is $a$, and moreover, we may assume that $M$ does not start with a type abstraction nor a type application.

2) If $\bar{E}$ erases to the empty context then $\bar{E}$ is a retyping context $R$.

3) If $R[M]$ is in $\iota$-normal form, then $R$ is of the form $\Lambda\vec{\alpha}$.[] $\bar{T}$.

The main helper lemma is:

**Lemma 28 (Inversion of type erasure)** Assume $[M] = a$

- If $a$ is $x$, then $M$ is of the form $R[x]$
- If $a$ is $c$, then $M$ is of the form $R[c]$
- If $a$ is $\lambda x. a_1$, then $M$ is of the form $R[\lambda x : \tau. M_1]$ with $[M_1] = a_1$
- If $a$ is $a_1 a_2$, then $M$ is of the form $R[M_1 M_2]$ with $[M_i] = a_i$

The proof is by an induction on $M$. 
4.4. TYPE ERASING SEMANTICS

Lemma 29 (Inversion of type erasure for well-typed values) Assume $\Gamma \vdash M : \tau$ and $M$ is $\iota$-normal. If $[M]$ is a value $v$, then $M$ is a value $V$. Moreover,

- If $v$ is $\lambda x. a_1$, then $V$ is $\Lambda \tilde{\alpha}. \lambda x : \tau. M_1$ with $[M_1] = a_1$.
- If $v$ is a partial application $c \, v_1 \ldots v_n$ then $V$ is $R[c \, \tilde{\tau} \, V_1 \ldots V_n]$ with $[V_i] = v_i$.

The proof is by induction on $M$. It uses the inversion of type erasure, then analysis of the typing derivation to restrict the form of retyping contexts. (Details of the proof p. 88)

Corollary 30 Let $M$ be a well-typed term in $\iota$-normal whose erasure is $a$

- If $a$ is $(\lambda x. a_1) \, v$ must be of the form $R[(\lambda x : \tau. M_1) \, V]$, with $[M_1]$ equal to $a_1$ and $[V]$ equal to $v$.
- If $a$ is a full application $c \, v_1 \ldots v_n$ then $M$ is of the form $R[c \, \tilde{\tau} \, V_1 \ldots V_n]$ with $[V_i] = v_i$. (Proof p. 88)

We may now prove backward simulation. It suffices to prove it when $M$ is $\iota$-normal. The general case follows, since one may first $\iota$-reduce $M$ to a normal form $M_0$, while preserving typings, thanks to subject reduction and type erasure; the lemma can then be applied to $M_0$ instead of $M$. Notice that this reasoning relies on the termination of $\iota$-reduction. Indeed, if $\iota$-reduction could diverge, it is unlikely that the semantics would be type erasing.

Termination of $\iota$-reduction follows indirectly from the termination of reduction in System $\mathcal{F}$. Its direct proof is also immediate, as $\iota$-reduction strictly decreases the number of type abstractions.

Proof (backward simulation): The proof is by induction on the reduction of $[M]$. We assume $M$ is in $\iota$-normal form.

Case $[M]$ is $(\lambda x. a_1) \, v$: By Corollary 30, $M$ is of the form $R[(\lambda x : \tau_1. M_1) \, V]$. Since $R$ is an evaluation context, $M$ reduces to $R[(x \mapsto V) \, M_1]$ whose erasure is $[x \mapsto v]a_1$, i.e. $a$.

Case $[M]$ is $e[a_1]$ and $a_1 \rightarrow a_2$: By Lemma 27, $M$ is of the form $E[M_1]$ where $[E]$ is $\tilde{e}$ and $[M_1]$ is $a_1$. By compositionality (Lemma 22), $M_1$ is well-typed. Since $M$ is $\iota$-normal and $E$ is an evaluation, $M_1$ is also $\iota$-normal. By induction hypothesis, $M_1$ reduces in one $\beta\delta$ step to a term $M_2$ whose erasure is $a_2$. Hence, by CONTEXT, $M$ reduces in one $\beta\delta$-step to the term $E[M_2]$ whose erasure is $e[a_2]$, i.e. $a$.

Case $[M]$ is a full application $(d \, v_1 \ldots v_n)$ and reduces to $a$: By Corollary 30, $M$ is of the form $R[M_0]$ where $M_0$ is $d \, \tilde{\tau} \, V_1 \ldots V_n$ a $[V_i]$ is $v_i$. Since $[M_0] \rightarrow a$, by the backward assumption for $\delta$-rules, there exists $M_0'$ such that $M_0 \rightarrow_\delta M_0'$ and $[M_0']$ is $a$. Let $M'$ be $R[M_0']$. Since $R$ is an evaluation context, we have $M \rightarrow_\delta M'$ and $[M']$ is $a$.

We may now easily transpose subject reduction and progress from the implicitly-typed version to the implicitly-typed version of System $\mathcal{F}$.
Theorem 12 (Type soundness for implicitly-typed System $F$)  
Progress and subject reduction holds in implicitly-typed System $F$.

\[ \text{Proof:} \text{ Assume that } \Gamma \vdash a_1 : \tau. \text{ By Lemma 24, there exists a term } M_1 \text{ such that } \Gamma \vdash M_1 : \tau. \text{ and } \lceil M_1 \rceil = a_1. \]

\[ \text{Progress:} \text{ Let } M_2 \text{ be the } \iota \text{-normal form of } M_1. \text{ By forward simulation, } \lceil M_2 \rceil \text{ is } a. \text{ By subject reduction, we have } \Gamma \vdash M_2 : \tau. \text{ By progress in } F, \text{ either } M_2 \beta\delta\text{-reduces and so does } a, \text{ by forward simulation (Lemma 25) or } M_2 \text{ is a value and so is its erasure } a_1 \text{ (by observation).} \]

\[ \text{Subject reduction:} \text{ Assume } a_1 \rightarrow a_2. \text{ By backward simulation (Lemma 26), there exists a term } M_2 \text{ such that } M_1 \rightarrow^* \beta\delta M_2 \text{ and } \lceil M_2 \rceil \text{ is } a_2. \text{ By subject reduction in } F, \text{ we have } \Gamma \vdash M_2 : \tau. \text{ By Lemma 24, we have } \Gamma \vdash a_2 : \tau, \text{ as expected.} \]

Remarks  The design of advanced typed systems for programming languages is usually done in explicitly-typed version, with a type-erasing semantics in mind, but this is not always checked in details (and sometimes not even made very clear). While the forward simulation is usually straightforward, the backward simulation is often harder. As the type system gets more complicated, reduction at the level of types also gets more involved. It is important and not always obvious that type reduction terminates and is rich enough to never block reductions that could occur in the type erasure.

For example, Crétin and Rémy (2012) extend System $F_\eta$ with abstraction over retyping functions, but keep the type systems bridled to preserve the type erasure semantics.

Bisimulation is a standard technique to show that compilation preserves the semantics given in small-step style. For example, it is heavily used in the CompCert project (Leroy, 2006) to prove the correctness of a compiler from C to assembly code, using the Coq proof assistant. The compilation from C to assembly code is decomposed into a chain of transformation using a dozen of successive intermediate languages; each of the transformation is then proved to be semantic preserving using bisimulation techniques.

4.5 Polymorphism and references

In this chapter, we have just shown how to extend simply-typed $\lambda$-calculus with polymorphism. In the previous chapter we have shown how to extend simply-typed $\lambda$-calculus with references. Can these extensions be combined together?

When adding references, we noted that type soundness relies on the fact that every reference cell (or memory location) has a fixed type. Otherwise, if a location had two types $\text{ref } \tau_1$ and $\text{ref } \tau_2$, one could store a value of type $\tau_1$ and read back a value of type $\tau_2$. Hence, it should also be unsound if a location could have type $\forall \alpha. \text{ref } \tau$ (where $\alpha$ appears in $\tau$) as it could then be specialized to both types $\text{ref } [\alpha \mapsto \tau_1] \tau$ and $\text{ref } [\alpha \mapsto \tau_2] \tau$. By contrast, a
location \( \ell \) can have type \( \text{ref}(\forall \alpha. \tau) \): this says that \( \ell \) stores values of polymorphic type \( \forall \alpha. \tau \), but \( \ell \), as a value, is viewed with the monomorphic type \( \text{ref}(\forall \alpha. \tau) \).

### 4.5.1 A counter example

Still, if System F is naively extended with references, it allows the construction of polymorphic references, which breaks subject reduction:

\[
\begin{align*}
\text{let } & y : \forall \alpha. \text{ref} (\alpha \to \alpha) = \Lambda \alpha. \text{ref} (\alpha \to \alpha) (\lambda z : \alpha. z) \quad (1) \\
& \text{in} \quad (y \text{ bool}) := (\text{bool} \to \text{bool}) \text{ not}; \quad (2) \\
& (! (\text{int} \to \text{int})(y (\text{int}))) 1 / \emptyset \quad (3) \\
& \not\! 1 / \ell \mapsto \text{not} \quad (4)
\end{align*}
\]

The program is well-typed, but reduces to the stuck expression “not 1”. So what went wrong? As described on the right-hand side, the fault is that the location is written at type \( \text{bool} \to \text{bool} \) and read back at type \( \text{int} \to \text{int} \). This is permitted because the location has a polymorphic type \( \forall \alpha. \text{ref} \alpha \to \alpha \). So this must be wrong. Indeed, the first reduction step uses the following rule (where \( V \) is \( \lambda x : \alpha. x \) and \( \tau \) is \( \alpha \to \alpha \)).

\[
\begin{array}{c}
\text{Context} \\
\text{ref } \tau \, V / \emptyset \rightarrow \ell / \ell \mapsto V
\end{array}
\]

While we have

\[
\begin{align*}
\alpha & \vdash \text{ref } \tau \, V / \emptyset : \text{ref } \tau \\
\alpha & \vdash \ell / \ell \mapsto V : \text{ref } \tau
\end{align*}
\]

We have

\[
\begin{align*}
\vdash \Lambda \alpha. \text{ref } \tau \, V / \emptyset : \forall \alpha. \text{ref } \tau & \quad \text{but not} \quad \vdash \Lambda \alpha. \ell / \ell \mapsto V : \forall \alpha. \text{ref } \tau
\end{align*}
\]

Hence, the context case of subject reduction breaks.

The typing derivation of \( \Lambda \alpha. \ell \) requires a store typing \( \Sigma \) of the form \( \ell : \tau \) and a derivation of the form (according to Rule \textbf{Loc} given below, page 4.5.2):

\[
\begin{array}{c}
\text{TABS} \\
\Sigma, \alpha \vdash \ell : \text{ref } \tau
\end{array}
\]

However, the typing context \( \Sigma, \alpha \) is ill-formed as \( \alpha \) appears free in \( \Sigma \). Instead, a well-formed premise should bind \( \alpha \) earlier as in \( \alpha, \Sigma \vdash \ell : \text{ref } \tau \), but then, Rule \textbf{TABS} cannot be applied.

By contrast, the expression \( \text{ref } \tau \, V \) is pure, so \( \Sigma \) may be empty:

\[
\begin{array}{c}
\text{TABS} \\
\alpha \vdash \text{ref } \tau \, V : \text{ref } \tau
\end{array}
\]

\[
\emptyset \vdash \Lambda \alpha. \text{ref } \tau \, V : \forall \alpha. \text{ref } \tau
\]
The expression $\Lambda\alpha.\ell$ is correctly rejected as ill-typed, so $\Lambda\alpha.\text{ref }\tau V$ should also be rejected. There is a fix to the bug known as this mysterious slogan:

One must not abstract over a type variable that might, after evaluation of the term, enter the store typing.

Indeed, this is what happens in our example. The type variable $\alpha$ which appears in the type of $V$ is abstracted in front of $\text{ref }\tau V$. When $\text{ref }\tau V$ reduces, $\alpha \rightarrow \alpha$ becomes the type of the fresh location $\ell$, which appears in the new store typing. This is all well and good, but how do we enforce this slogan?

In the context of ML, a number of rather complex historic approaches have been followed: see Leroy (1992) for a survey. Then came Wright (1995), who suggested an amazingly simple solution, known as the value restriction: only value forms can be abstracted over.

The problematic proof case vanishes, as we now never reduce under type abstraction. The form $\Lambda\alpha.E$ of evaluation context becomes useless and can be removed. Subject reduction holds again. Let us prove it.

### 4.5.2 Internalizing configurations

A configuration $M / \mu$ is an expression $M$ in a memory $\mu$. Intuitively, the memory can be viewed as a recursive extensible mutable record. The configuration $M / \mu$ may be viewed as the recursive definition (of values) $\text{let rec } m : \Sigma = \mu \text{ in } [\ell \mapsto m.\ell]M$ where $\Sigma$ is a store typing for $\mu$. The store typing rules are coherent with this view. For instance, allocation of a reference is a reduction of the form:

$$\text{let rec } m : \Sigma = \mu \text{ in } E[\text{ref }\tau V] \rightarrow \text{let rec } m : \Sigma, \ell : \tau = \mu, \ell \mapsto v \text{ in } E[m.\ell]$$

For this transformation to preserve well-typedness, it is clear that the evaluation context $E$ must not bind any type variable appearing in $\tau$; otherwise, we are violating the scoping rules.

Let use clarify the typing rules for configurations:

$$\text{let rec } m : \Sigma = \mu \text{ in } E[\text{ref }\tau]$$

Because we explicitly introduce type variables in judgments, closed configurations must be typed in an environment composed of type variables. Because we never reduce under type abstraction, these variables need not be changed during evaluation and can be placed in front of the store typing.
Judgments are now of the form \( \vec{\alpha}, \Sigma, \Gamma \vdash M : \tau \) although we may see \( \vec{\alpha}, \Sigma, \Gamma \) as a whole typing context \( \Gamma' \). For locations, we need a new context formation rule:

\[
\frac{\text{WFEnvLoc}}{\vdash \Gamma \quad \Gamma \vdash \tau \quad \ell \notin \text{dom}(\Gamma')}
\]

\( \vdash \Gamma, \ell : \tau \)

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form \( \vec{\alpha}, \Sigma, \Gamma \) where \( \Sigma \) only binds locations (to arbitrary types) and \( \Gamma \) does not bind locations.

The typing rule for memory locations (where \( \Gamma \) is of the form \( \vec{\alpha}, \Sigma, \Gamma' \)) is:

\[
\frac{}{\Gamma \vdash \ell : \text{ref } \Gamma(\ell)}
\]

In System F, typing rules for references need not be primitive. We may instead treat them as constants of the following types:

\[
\text{ref} : \forall \alpha. \alpha \to \text{ref } \alpha \quad (!) : \forall \alpha. \text{ref } \alpha \to \alpha \quad (=:) : \forall \alpha. \text{ref } \alpha \to \alpha \to \text{unit}
\]

They are all destructors (event \( \text{ref} \)) with the obvious arities.

The \( \delta \)-rules are adapted to carry explicit type parameters:

\[
\begin{align*}
\text{ref } \tau V / \mu & \rightarrow \ell / \mu[\ell \mapsto V] \quad \text{if } \ell \notin \text{dom}(\mu) \\quad \ell := (\tau) V / \mu & \rightarrow () / \mu[\ell \mapsto V] \quad !\tau \ell / \mu & \rightarrow \mu(\ell) / \mu
\end{align*}
\]

Type soundness can now be stated as

**Lemma 31** \( \delta \)-rules preserve well-typedness of closed configurations.

**Theorem 13 (Subject reduction)** Reduction of closed configurations preserves well-typedness.

**Lemma 32** A well-typed closed configuration \( M/\mu \) where \( M \) is a full application of constants \( \text{ref}, (!), \) and \((=)\) to types and values can always be reduced.

**Theorem 14 (Progress)** A well typed irreducible closed configuration \( M/\mu \) is a value.

As a sanity check, the problematic program is now syntactically ill-formed:

\[
\lambda \alpha. \text{ref } (\alpha \to \alpha) = \Lambda \alpha. \text{ref } (\alpha \to \alpha) \quad (\lambda z : \alpha. z) \quad \text{in}
\]

\[
(\text{y bool}) := (\text{bool} \to \text{bool}) \quad \text{not};
\]

\[
(! \text{(int} \to \text{int}) \quad (y \text{(int)}) \quad 1
\]

Indeed, \( \text{ref } (\alpha \to \alpha) \quad (\lambda z : \alpha. z) \) is not a value, but the application of a unary destructor to a value, so the expression \( \Lambda \alpha. \text{ref } (\alpha \to \alpha) \quad (\lambda z : \alpha. z) \) is not allowed.
Consequences  With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references. This style of introducing references in System F (or in ML) is not a conservative extension.

Assuming functions map and id of respective types \( \forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha \) and \( \forall \alpha. \alpha \rightarrow \alpha \), the expression \( \Lambda \alpha \cdot \text{map } \alpha \ (\text{id } \alpha) \) is now ill-typed. A common work-around is to perform a manual \( \eta \)-expansion \( \Lambda \alpha \cdot \lambda y : \text{list } \alpha. \text{map } \alpha \ (\text{id } \alpha) \ y \). However, in the presence of side effects, \( \eta \)-expansion is not semantics preserving, so this must not be done blindly.

In practice, the value restriction can be slightly relaxed by enlarging the class of value-forms to a syntactic category of so-called non-expansive terms—terms whose evaluation will definitely not allocate new reference cells. Non-expansive terms form a strict superset of value-forms. Garrigue (2004) relaxes the value restriction in a more subtle way, which is justified by a subtyping argument. For instance, the following expressions may be well-typed:

- \( \Lambda \alpha. ((\lambda x : \tau. U) U) \) because the inner expression is non-expansive;
- \( \Lambda \alpha. (\text{let } x : \tau = U \text{ in } U) \), which is its syntactic sugar;
- \( \text{let } x : \forall \alpha. \text{list } \alpha = \Lambda \alpha. (M_1 M_2) \text{ in } M \) because \( \alpha \) appears only positively in the type of \( \text{eapp} M_1 M_2 \).

OCaml implements both refinements.

In fact, \( \Lambda \alpha. M \) need only be forbidden when \( \alpha \) appears negatively in the type of some exposed expansive terms where exposed subterms are those that do not appear under some \( \lambda \)-abstraction. For instance, the expression

\[
\text{let } x : \forall \alpha. \text{int} \times (\text{list } \alpha) \times (\alpha \rightarrow \alpha) = \Lambda \alpha. (\text{ref} \ (1 + 2), (\lambda x : \alpha. x) \text{ Nil}, \lambda x : \alpha. x) \text{ in } M
\]

may be well-typed because \( \alpha \) appears only in the type of the non-expansive exposed expressions \( \lambda x : \alpha. x \) and positively in the type of expansive expression \( (\lambda x : \alpha. x) \text{ Nil} \).

(This refinement is not implemented in OCaml, though.)

Remark  Experience has shown that the value restriction is tolerable. Even though it is not conservative, the search for better solutions has been pretty much abandoned.

In a type-and-effect system (Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994), or in a type-and-capability system (Charguéraud and Pottier, 2008), the type system indicates which expressions may allocate new references, and at which type. There, the value restriction is no longer necessary—but these systems are heavy. However, if one extends a type-and-capability system with a mechanism for hiding state, which remains useful even in those systems, the need for the value restriction re-appears.

Pottier and Protzenko (2012) are designing a language Mezzo where mutable states is tracked quite precisely, with permissions, ownership, linear types that even enable a reference to even change the type of its values over time, which is called strong update.
4.6 Damas and Milner’s type system

Damas and Milner’s type system [Milner (1978)] offers a restricted form of polymorphism, while avoiding the difficulties associated with type inference in System F. This type system is at the heart of Standard ML, OCaml, and Haskell.

The idea behind the definition of ML is to make a small extension of simply-typed \( \lambda \)-calculus that enables to factor out several occurrences of the same subexpression \( a_1 \) in a term of the form \([x \mapsto a_1]a_2\) using a let-binding form let \( x = a_1 \) in \( a_2 \) so as to avoid code duplication.

Expressions of the simply-typed \( \lambda \)-calculus are extended with a primitive let-binding, which can also be viewed as a way of annotating some redexes \( (\lambda x. a_2) a_1 \) in the source program. This actually provides a simple intuition behind Damas and Milner’s type system: a closed term has type \( \tau \) if and only if its let-normal form has type \( \tau \) in simply-typed \( \lambda \)-calculus. A term’s let-normal form is obtained by iterating the following rewrite rule, in any context:

\[
\text{let } x = a_1 \text{ in } a_2 \quad \rightarrow \quad a_1;[x \mapsto a_1]a_2
\]

Notice that we use a sequence starting with \( a_1 \) and not just \([x \mapsto a_1]a_2\). This is to enforce well-typedness of \( a_1 \) in the pathological case where \( x \) does not appear free in \( a_2 \). If we disallow this pathological case (e.g. well-formedness could require that \( x \) always occurs in \( a_2 \)) then we could just use the more intuitive rewrite rule:

\[
\text{let } x = a_1 \text{ in } a_2 \quad \rightarrow \quad [x \mapsto a_1]a_2
\]

This intuition suggests type-checking and type inference algorithms. However, these algorithms are not practical, because they have intrinsic exponential complexity; and separate compilation prevents reduction to let-normal forms.

In the following, we study a direct presentation of Damas and Milner’s type system, which does not involve let-normal forms. It is practical, because it leads to an efficient type inference algorithm (presented in chapter §5); and it supports separate compilation.

4.6.1 Definition

The language ML is usually presented in its implicitly-typed version, where terms are given by:

\[
a ::= x \mid c \mid \lambda x. a \mid a \ a \mid \text{let } x = a \text{ in } a \ldots
\]

The let construct is no longer sugar for a \( \beta \)-redex but a primitive form that will be typed especially.

The language of types lies between those for simply-typed \( \lambda \)-calculus and System F; it is stratified between types and type schemes. The syntax of types is that of simply-typed \( \lambda \)-calculus, but a separate category of type schemes is introduced:

\[
\tau ::= \alpha \mid \tau \rightarrow \tau \mid \ldots \quad \sigma ::= \tau \mid \forall \alpha. \sigma
\]
CHAPTER 4. POLYMORPHISM AND SYSTEM F

All quantifiers must appear in *prenex position*, so type schemes are less expressive than System-F types. We often write \( \forall \alpha. \tau \) as a short hand for \( \forall \alpha_1, \ldots, \forall \alpha_n. \tau \). When viewed as a subset of System F, one must think of *type schemes* are the primary notion of types, of which *types* are a subset.

An ML typing context \( \Gamma \) binds program variables to *type schemes*. In the implicitly-typed presentation, type variables are often introduced implicitly and not part of \( \Gamma \). However, we keep below the equivalent presentation where type variables are declared in \( \Gamma \). Judgments now take the form \( \Gamma \vdash a : \sigma \). Types form a subset of type schemes, so type environments and judgments can contain types too.

The standard, non-syntax-directed presentation of ML is given in Figure 4.3. Rule Let moves a type scheme into the environment, which Var can exploit. Rule Abs and App are unchanged. \( \lambda \)-bound variables receive a *monotype*. Rule Gen and Inst are as in implicitly-typed System F, except that *type variables are instantiated with monotypes*.

For example, here is a type derivation that exploits polymorphism (writing \( \Gamma \) for \( f : \forall \alpha. \alpha \rightarrow \alpha \)) for an implicitly-typed term (omitting the IML- prefix of typing rules):

Notice that Rule Gen is used above Let (on the left-hand side), and Inst is used below Var. In fact, we will see below that every type derivation can be transformed into one of this form.

As a counter-example, the term \( \lambda f. (f \, 0, \, f \, \text{true}) \) is *ill-typed*. Indeed, as it contains no “let” construct, it is type-checked exactly as in simply-typed \( \lambda \)-calculus, where it is ill-typed, because \( f \) must be assigned a type \( \tau \) that must simultaneously be of the form \( \text{int} \rightarrow \tau_1 \) and \( \text{bool} \rightarrow \tau_2 \), but there is no such type. Recall that this term is well-typed in implicitly-typed System F because \( f \) can be assigned, for instance, the polymorphic type \( \forall \alpha. \alpha \rightarrow \alpha \).
4.6. DAMAS AND MILNER’S TYPE SYSTEM

![Typing rules for eML (explicitly-typed ML)](image)

While we rather use implicitly-typed terms in programs, we usually prefer to use an explicitly-typed presentation of ML in proofs. We thus identify a subset of terms of System F whose type erasure coincide with terms of ML. The subset of terms is defined by the follow syntax:

\[ M \in \text{eML} := x | c | \lambda x: \tau. M | M M | \Lambda \alpha. M | M \tau | \text{let } x: \sigma = M \text{ in } M \ldots \]

where \( \tau \) and \( \sigma \) are ML-types and type schemes and not arbitrary System-F types. The typing rules for explicitly-typed terms are given on Figure 4.4.

These are restrictions of the typing rules of System-F to terms and types of ML. Therefore, if \( \Gamma \vdash_{\text{eML}} M : \sigma \) then \( \Gamma \vdash_{F} M : \sigma \). In particular, explicitly-typed terms of ML have unique typing derivations—and actually unique types—as in System-F.

Unfortunately, the converse is not true—when \( M \) is syntactically in ML and \( \Gamma \) and \( \sigma \) are well-formed in eML, of course. Hence, the relation \( \vdash_{\text{eML}} \) cannot be defined as the restriction of \( \vdash_{F} \) to ML environments terms and type schemes.

**Exercise 34** Find a term \( M \) that is syntactically in eML and a type scheme \( \sigma \) such that \( \Gamma \vdash_{F} M : \sigma \) holds but \( \Gamma \vdash_{\text{eML}} M : \sigma \) does not hold. (Solution p. 89)

### 4.6.2 Syntax-directed presentation

Explicitly-typed terms of ML have unique typing derivations—and actually unique types—as in System-F. By contrast with explicitly-typed terms, implicitly-typed terms have several types, since parameters of functions are not annotated, but also several typing derivations, since places for type abstraction and type applications are not specified either, much as in System F.

Interestingly, there is a syntax-directed presentation of implicitly-typed ML terms where the shape of typing derivations is entirely determined by the term and is thus unique. Taking the explicitly-typed view, this amounts to restricting the source terms so that there is no choice for placing type abstraction and type applications.
Let $x\text{ML}$ be the subset of explicitly-typed $\text{ML}$ defined by the following grammar

$$
\begin{align*}
N \in x\text{ML} & \quad ::= \quad \Lambda \bar{\alpha}.Q \\
Q & \quad ::= \quad x \bar{\tau} \mid Q Q \mid \lambda x:\tau. Q \mid \text{let } x : \sigma = N \text{ in } Q
\end{align*}
$$

where $\tau$ here ranges over simple types and such that all type variables are fully instantiated. That is, we request that the arity of $\bar{\tau}$ in $x \bar{\tau}$ be the arity of $\bar{\alpha}$ in the type scheme $\forall \bar{\alpha}. \tau$ assigned to the variable $x$. In particular, all $Q$-terms are typed with simple types.

Specializing the typing rules of $e\text{ML}$ (Figure 4.4) to the syntax of $x\text{ML}$ gives the typing rules of $x\text{ML}$ on Figure 4.5. By construction, terms of $x\text{ML}$ are a syntactic subset of terms of $e\text{ML}$. By construction, we also have if $\Gamma \vdash_{x\text{ML}} M : \sigma$ then $\Gamma \vdash_{e\text{ML}} M : \sigma$.

Conversely, we wish to show that any term $M$ typable in $e\text{ML}$ can be mapped to a term $N$ typable in $x\text{ML}$ that has the same type erasure. For this purpose, we define on Figure 4.6 a normalization judgment $\Gamma \vdash M : \sigma \Rightarrow N$ by inference rules, which can also be read as an algorithm that performs:

- Type $\eta$-expansion of every occurrence of a variable according to the arity of its type scheme (Rule $\text{VAR}$). This ensures that every occurrence of a type variable will be fully specialized—hence assigned a monomorphic type.
4.6. DAMAS AND MILNER’S TYPE SYSTEM

The translation is well-defined for all $e\text{ML}$ terms, since it follows the structure of the typing derivation in $e\text{ML}$. Formally, if $\Gamma \vdash e\text{ML} M : \sigma$ holds then $\Gamma \vdash x : \sigma \Rightarrow N$ holds. The proof is by induction on $M$ and all cases are obvious.

Moreover, if $\Gamma \vdash M : \sigma$ holds, then $\Gamma \vdash s\text{ML} N : \sigma$ also holds and $M$ and $N$ have the same erasure. The proof is also by induction on $M$. The preservation of erasure is immediate. The only non obvious cases for well-typedness of $N$ are $\text{Norm-Tapp}$, which performs strong $\iota$-reduction and uses type substitution (Lemma 21), and $\text{Norm-Let}$, which extrudes type abstractions.

Another way to look at the normalization of terms is as a rewriting of the typing derivations so that all applications of $\text{Inst}$ come immediately after $\text{Var}$ and all applications of $\text{Gen}$ come immediately above rule $\text{Let}$ or at the bottom of the derivation—as imposed by the grammar of $x\text{ML}$ terms where $Q$-terms can only have monomorphic types.

In summary, any term of $e\text{ML}$ can be rearranged as a term of $x\text{ML}$ with the same type erasure. By dropping type information in terms of $x\text{ML}$, we then obtain a syntax-directed presentation of implicitly-typed $\text{ML}$, called $s\text{ML}$:

Then, the judgments $\Gamma \vdash e\text{ML} a : \tau$ and $\Gamma \vdash s\text{ML} a : \tau$ are equivalent.

However, for type inference, we rather use the equivalent presentation in Figure 4.7 called $i\text{ML}$ (or the inference type system) where type variables are not explicitly declared in the typing context—hence, the side condition for generalization on rule $\text{Let}$.

In this final system, type substitution (Lemma 21), which we will use for type inference, can be restated as follows:

- Strong $\iota$-reduction, i.e. type $\beta$-reduction (Rule [TAPP]): this cancels type applications of type abstractions. As a result, elaborated terms do not contain any $\iota$-redex.
Lemma 33 (Type Substitution) Typings are stable by substitution. If $\Gamma \vdash a : \tau$ then $\varphi \Gamma \vdash a : \varphi \tau$. for any substitution $\varphi$.

4.6.3 Type soundness for ML

Since ML is a subset of $F$, which has been proved sound, we know that ML is sound, i.e. that ML programs cannot go wrong. This also implies that progress holds in ML. However, we do not know whether subject reduction holds for ML. Indeed, ML expressions could reduce to System F expressions that are not in the ML subset. Most proofs of subject reduction for implicitly-typed ML work directly with implicitly-typed terms. See for instance (Wright and Felleisen, 1994; Pottier and Rémy, 2005).

Subject-reduction in $\varepsilon$ML The proof of subject reduction follows the same schema as for System F (Theorem 9). The main part of the proof works almost unchanged. However, it uses auxiliary lemmas (inversion, permutation, weakening, type substitution, term substitution, compositionality) that all need to be rechecked, since those lemmas conclude with typing judgments in $F$ that may not necessarily hold in $\varepsilon$ML. Unsurprisingly, all proofs can be easily adjusted.

An indirect proof reusing subject-reduction in System F We also present an indirect proof that reuses subject reduction and progress in System F and the syntax-directed presentation of ML.

To establish subject-reduction in ML, let $a_1$ be an implicitly-typed ML term such that both $\bar{\alpha} \vdash_{\text{ML}} a_1 : \sigma$ and $a_1 \rightarrow a_2$ hold. There exists an explicitly-typed term $M_1$ such that $\bar{\alpha} \vdash_{\text{eML}} M_1 : \sigma$ and $\left[ M_1 \right] = a_1$. By normalization, we may elaborate $M_1$ into a term $N_1$ of xML such that $\bar{\alpha} \vdash_{\text{xML}} N_1 : \sigma$ and the $\left[ N_1 \right] = \left[ M_1 \right]$. Moreover, $N_1$ is by construction $\iota$-normal. Since $\text{xML}$ is a subset of System F, we have $\bar{\alpha} \vdash_F N_1 : \sigma$. By backward simulation in System F (Lemma 26), there exists $N_2$ in F whose type erasure is $a_2$ and such that $N_1 \rightarrow_{\beta} N_2$ (since $N_1$ is $\iota$-normal). We show below that there exists a strong $\iota$-reduction $M_2$ of $N_2$ that is in $\text{xML}$ and such that $\bar{\alpha} \vdash_{\text{xML}} N_2 : \sigma$. Therefore, we have $\bar{\alpha} \vdash_{\text{eML}} M_2 : \sigma$ and since the type erasure of $M_2$ is that of $N_2$, i.e. $a_2$, we have $\bar{\alpha} \vdash_{\text{ML}} a_2 : \sigma$, as expected.

It thus remains to check that given a term $N_1$ such that $\Gamma \vdash_{\text{ML}} N_1 : \sigma$ and $N_1 \rightarrow_{\beta} N_2$, there exists a term $M_2$ in $\text{xML}$ that is a strong $\iota$-reduction of $N_2$ and such that $\Gamma \vdash_{\text{xML}} N_2 : \sigma$. This can be decomposed into the existence of $M_2$ and type preservation by strong $\iota$-reduction.

The $\beta$-reduction step may occur in any evaluation context and is one of two forms. If it is a normal $\beta$-reduction:

$$(\lambda x : \tau. Q) \ V \rightarrow [x \mapsto V]Q$$

it preserves syntactic membership in $\text{eML}$, because since $x$ is bound to a type and its occurrences in $M$ cannot be specialized. However, if it is a let-reduction

\begin{align*}
\text{let } x : \forall \alpha. \tau & = V \text{ in } Q \rightarrow [x \mapsto V]Q
\end{align*}
then occurrences of $x$ in $Q$, which are of the form $x\overline{\tau}$, become $V\overline{\tau}$ and may contain $\nu$-redexes—which are not allowed in $x\text{ML}$. Fortunately, $V$ is necessarily of the form $\Lambda\overline{\alpha}.V'$ where the arity of $\overline{\alpha}$ is equal to that of $\overline{\tau}$. Hence, we may immediately perform a sequence of $\nu$-reduction that brings the term back into $x\text{ML}$ and in $\nu$-normal form. Notice however that this $\nu$-redex is not in general in a call-by-value evaluation context. Indeed, $x$ may appear under an abstraction in $M$. Hence, this is a strong reduction step.

For type reduction, we need to ensure that strong $\nu$-reduction is also type-preserving. This is an easy auxiliary proof—but not a consequence of subject reduction, which we have only proved for reduction in call-by-value evaluation contexts.
4.7 Ommitted proofs and answers to exercises

Solution of Exercise 24

As in the case where pairs are primitive, we introduce one constructor \( (\cdot, \cdot) \) of arity 2 and and two destructors \( \text{proj}_1 \) and \( \text{proj}_2 \) of arity 1, with the following types in \( \Delta \)

\[
\text{Pair} : \forall \alpha_1, \forall \alpha_2. \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2 \\
\text{proj}_i : \forall \alpha_1, \forall \alpha_2. \alpha_1 \times \alpha_2 \rightarrow \alpha_i
\]

and the two reduction rules:

\[
\text{proj}_i \tau_1 \tau_2 (\text{Pair} \tau_1' \tau_2' V_1 V_2) \rightarrow V_i \quad (\delta_i)
\]

We then only need to verify that \( \delta \) preserves types and ensure progress.

Case Type preservation: Assume that \( \Gamma \vdash \text{proj}_i \tau_1 \tau_2 (\text{Pair} \tau_1' \tau_2' V_1 V_2) : \tau \). By inversion, it must be the case that \( \tau \) is equal to \( \tau_i \) and \( \Gamma \vdash V_i : \tau_i \) holds, which ensures our goal \( \Gamma \vdash V_i : \tau \).

Case Progress: Assume that \( \Gamma \vdash M : \tau \) and \( M \) is of the form \( \text{proj}_i \tau_1 \tau_2 V \). By the inversion lemma, \( \tau \) must be a product type \( \tau_1 \times \tau_2 \) such that \( \Gamma \vdash V : \tau_1 \times \tau_2 \). By the classification lemma, \( V \) must be a pair, i.e. of a form \( \text{Pair} \tau_1 V_1 V_2 \). Hence, \( M \) reduces to \( V_i \) by \( \delta_i \).

Solution of Exercise 25

We introduce a new type constructor \( \text{bool} \), two nullary constructors \( \text{true} \) and \( \text{false} \) of type \( \text{bool} \) and one ternary destructor \( \text{ifcase} \) of type \( \forall \alpha. \text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \) with two reduction rules:

\[
\text{ifcase} \tau \text{true} V_1 V_2 \rightarrow V_1 \quad \text{ifcase} \tau \text{false} V_1 V_2 \rightarrow V_2
\]

This extension is sound.

However, it defines a strict semantics for the conditional, while a lazy semantics is expected: indeed, since the destructor is ternary, \( \text{ifcase} \tau V_0 [] M \) and \( \text{ifcase} \tau V_0 V_1 [] \) are evaluations contexts, which allows to reduce the two branches before selecting the right one.

An easy fix is to introduce \( \text{iflazy} \tau M_0 M_1 M_2 \) as syntactic sugar for

\[
(\text{ifcase} \tau M_0 (\lambda(): \text{unit}) M_1) (\lambda(): \text{unit}) M_2)) ()
\]

and exposing it to the user, while hiding the primitive \( \text{ifcase} \) from the user.

Solution of Exercise 26

We introduce a new unary type constructor \( \text{list} \); two constructors \( \text{Nil} \cdot \) and \( \text{Cons} \) of types \( \forall \alpha. \text{list} \alpha \) and \( \forall \alpha. \alpha \rightarrow \text{list} \alpha \rightarrow \text{list} \alpha \); and one destructor \( \text{matchlist} \cdot \cdot \cdot \cdot \) of type:

\[
\forall \alpha \beta. \text{list} \alpha \rightarrow \beta \rightarrow (\alpha \rightarrow \text{list} \alpha \rightarrow \beta) \rightarrow \beta
\]
4.7. OMITTED PROOFS AND ANSWERS TO EXERCISES

with the two reduction rules:

\[
\text{matchlist } \tau(\text{Nil } \tau') V_n V_c \longrightarrow V_n \\
\text{matchlist } \tau(\text{Cons } \tau' V_h V_t) V_n V_c \longrightarrow V_c V_h V_t
\]

Solution of Exercise 27

In ML, we may define the datatype:

\[
\text{type any } = \text{Fold of (any } \rightarrow \text{ any)}
\]

This can be simulated by adding a new type \text{any}, a constructor \text{Any} and a destructor \text{unany} of types \text{(any } \rightarrow \text{ any)} \rightarrow \text{any} and \text{any} \rightarrow \text{(any } \rightarrow \text{ any)}, respectively, with the following reduction rule:

\[
\text{unfold (Fold } V \text{)} \longrightarrow V
\]

\(\delta_{\text{any}}\)

Let us check soundness of this extension:

Case Type preservation: Assume that \(\Gamma \vdash \text{unfold (Fold } V \text{)} : \tau\). By inversion, we known that \(\tau\) is \text{any } \rightarrow \text{ any} and that \(\Gamma \vdash V : \text{any } \rightarrow \text{ any}\), which shows our goal \(\Gamma \vdash V : \tau\).

Case Progress: Assume that \(\Gamma \vdash \text{unfold } V : \tau\). By inversion, \(\tau\) must be \text{any } \rightarrow \text{ any} and \(\Gamma \vdash V : \text{any}\) holds. By classification, \(V\) must be \text{Fold } V_0.\) Hence, \text{unfold } V reduces.

The fixpoint can be defined in the \(\lambda\)-calculus (or in ML with recursive types) as:

\[
\text{let } zfix g = (\text{fun } x \rightarrow x) (\text{fun } z \rightarrow g (\text{fun } v \rightarrow z z v))
\]

We may implement \text{zfix} in ML without recursive types as:

\[
\text{type any } = \text{Fold of (any } \rightarrow \text{ any)};;
\text{let unfold (Fold } x \text{)} = x;;
\text{let } zfix g =
(\text{fun } x \rightarrow \text{unfold (x (Fold } x)))
(\text{fun } z \rightarrow \text{Fold (g (fun } v \rightarrow \text{unfold ((unfold } z) z v)))};;
\]

Proof of Lemma 20

Assume \(\Gamma, x : \tau_0, \Gamma' \vdash M : \tau\) (1) and \(\Gamma \vdash M_0 : \tau_0\) (2). We show \(\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau\) (3). by induction and cases on \(M\) and applying the inversion lemma to (1).

Case \(M\) is \(x\): By (1), it must be the case that \(\tau\) is equal to \(\tau_0\). Hence, the goal (3) is \(\Gamma, \Gamma' \vdash M_0 : \tau_0\), which follows from the hypothesis (2) by weakening.

Case \(M\) is \(y\) when \(y \notin x\): By (1), \(y : \tau\) is in \text{dom}(\Gamma, x : \tau_0, \Gamma') , actually in \text{dom}(\Gamma, \Gamma') , since \(y\) is not \(x\). Hence the goal (3) follows by Rule VAR

Case \(M\) is \(c\): By (1), \(c : \tau\) is in \(\Delta\). Hence, the goal (3) follows by Rule VAR
Case $M$ is $\lambda y : \tau_1. M_1$: By (1), $\tau$ is of the form $\tau_2 \rightarrow \tau_1$ and $\Gamma, x : \tau_0, \Gamma', y : \tau_2 \vdash M_1 : \tau_1$ holds. By induction hypothesis, we have $\Gamma, \Gamma', y : \tau_2 \vdash [x \mapsto M_0]M_1 : \tau_1$. By rule [ABS] we have $\Gamma, \Gamma' \vdash \lambda y : \tau_2. [x \mapsto M_0]M_1 : \tau_1$, which is the goal (3).

Case $M$ is $\Lambda \alpha. M_1$: By (1), we have $\Gamma, x : \tau, \Gamma', \alpha \vdash M_1 : \tau_1$ and $\tau$ is equal to $\forall \alpha. \tau_1$. By induction hypothesis, we have $\Gamma, \Gamma', \alpha \vdash [x \mapsto M_0]M_1 : \tau_1$. By rule [TABS] we have $\Gamma \vdash \Lambda \alpha. [x \mapsto M_0]M_1 : \forall \alpha. \tau_1$, which is the goal (3).

Case $M$ is $M_1 M_2$ or $M$ is $M_1 \tau_1$: Immediate.

**Proof of Lemma 21**

The proof is by induction on $M$ using inversion of the typing derivation of $\Gamma, \alpha, \Gamma' \vdash M : \tau$ (1). We write $\theta$ for $[\alpha \mapsto \tau]$.

Case $M$ is $x$: By (1), we have $x : \tau$ must be in $\Gamma, \alpha, \Gamma'$. If $x : \tau$ is in $\Gamma$, then by well-formedness of types, $\alpha$ does not appear free in $\tau$. Hence $\theta \tau$ is $\tau$ and $x : \theta \tau$ is in $\Gamma$. Otherwise, $x : \tau$ is in $\Gamma'$ and $x : \theta \tau$ is in $\theta \Gamma'$. In both cases, $x : \theta \tau$ is in $\Gamma, \theta \Gamma'$. Hence, the conclusion follows by Rule [VAR]

Case $M$ is $c$: By (1), we have $c : \tau$ is in $\Delta$ and $\tau$ is closed. Hence $\theta \tau$ is equal to $\tau$ and $c : \theta \tau$ is still in $\Delta$. Thus, the conclusion follows by Rule [CONST]

Case $M$ is $\lambda x : \tau_0. M_1$: By (1), we have $\Gamma, \alpha, \Gamma', x : \tau_0 \vdash M_1 : \tau$. By induction hypothesis, $\Gamma, \theta(\Gamma', x : \tau_0) \vdash M_1 : \tau$, i.e. $\Gamma, \theta \Gamma', x : \theta \tau_0 \vdash \theta M_1 : \theta \tau$. By Rule [ABS] we have $\Gamma, \theta \Gamma' \vdash \lambda x : \theta \tau_0. \theta M_1 : \theta \tau$, i.e. $\Gamma, \theta \Gamma' \vdash \theta(\lambda x : \tau_0. M_1) : \theta \tau$.

Case $M$ is $\Lambda \beta. M_1$: By (1), we have $\Gamma, \alpha, \Gamma', \beta \vdash M_1 : \tau$. By induction hypothesis, we have $\Gamma, \theta(\Gamma', \beta) \vdash \theta M_1 : \theta \tau$, which is equal to $\Gamma, \theta \Gamma', \beta \vdash \theta M_1 : \theta \tau$. By rule [TABS] we $\Gamma, \theta \Gamma' \vdash \Lambda \beta. \theta M_1 : \theta \tau$, which is equal to $\Gamma, \theta \Gamma' \vdash \theta(\Lambda \beta. M_1) : \theta \tau$.

Case $M$ is $M_1 M_2$ or $M$ is $M_1 \tau_1$: Immediate.

**Solution of Exercise 30**

Take, for instance, $\lambda f. \lambda x. \lambda y. (f y, f x)$ for $a_1$ (notice the inverse order of fields in the pair) and $\lambda f. \lambda x. \lambda y. f (f x), f (f y)$ for $a_2$.

**Solution of Exercise 31**

Choose, for instance,

$$\Lambda \alpha_1. \Lambda \alpha_2. \Lambda \varphi_1. \Lambda \varphi_2. (\forall \alpha. \varphi_1(\alpha) \rightarrow \varphi_2(\alpha)) \rightarrow \varphi_1(\alpha_1) \rightarrow \varphi_1(\alpha_2) \rightarrow \varphi_2(\alpha_1) \times \varphi_2(\alpha_1)$$

for $\tau_0$. We recover $\tau_1$ by choosing the constant functions $\lambda \alpha. \alpha_i$ for $\varphi_i$ and $\tau_2$ by choosing the identity $\lambda \alpha. \alpha$ for both $\varphi_1$ and $\varphi_2$.
4.7. OMITTED PROOFS AND ANSWERS TO EXERCISES

Solution of Exercise 32

1) Both directions follow from rule \texttt{INST-GEN}, just applying the substitution \(\alpha \mapsto \alpha\) for the direct implication and just generalizing over \(\alpha\) for the reverse.

2) Rule \texttt{Distrib-Right} is a particular case of \texttt{Distributivity}, indeed. Assuming \(\alpha \notin \text{ftv}(\tau_1)\), and using the previous equivalence (1), we have

\[
\frac{\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2)}{\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2)}
\]

Conversely, we have the following derivation:

\[
\frac{\text{Inst-Gen}}{\forall \alpha. \tau_1 \leq \tau_1}
\]
\[
\frac{\text{Congruence-} \rightarrow}{\tau_1 \rightarrow \tau_2 \leq (\forall \alpha. \tau_1) \rightarrow \tau_2}
\]
\[
\frac{\text{Congruence-} \rightarrow}{\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq \forall \alpha. (\forall \alpha. \tau_1) \rightarrow \tau_2}
\]
\[
\frac{\text{Distrib-Right}}{\forall \alpha. (\tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2)}
\]

Solution of Exercise 33

We extend the \(\lambda\)-calculus with a binary constructor \(\text{Pair}\) and two unary destructors \(\text{proj}_i\) for \(i \in \{1, 2\}\) with the \(\delta\)-rules:

\[
\text{proj}_i (\text{Pair} \, v_1 \, v_2) \longrightarrow \delta \, v_i
\]

The reduction \(\text{proj}_1 (\text{Pair} \, v \, (\lambda x. \text{Pair} \, \text{Pair})) \longrightarrow \delta \, v\) is correct, even though the right component of the pair is ill-typed, hence \(\delta\)-reduction is larger than the type-erasure of \(\delta\)-reduction on explicitly typed terms. Still, it contains it (forward simulation); and it does not contain more (backward simulation) when we restrict to well-typed expressions. Both cases are really easy:

Proof: Let \(M\) be of the form \(\text{proj}_i \, \tau_1 \, \tau_2 \, V_0\) such that \(\Gamma \vdash M : \tau\). By inversion of typing rules, \(\Gamma \vdash V_0 : \tau_1 \times \tau_2\). By the classification lemma, \(V_0\) is of the form \(\text{Pair} \, \tau_1 \, \tau_2 \, V_1 \, V_2\). Observe that \(M\) reduces to \(V_i\) (1); and \([M] = \text{proj}_i \, (\text{pair} \, [V_1] \, [V_2])\) which reduces to \([V_i]\) (2).

Case forward: Assume \(M \longrightarrow \delta \, M'\). Then, \(M'\) is \(V_i\) and by (2), \([M] \longrightarrow \delta \, [V_i]\).

Case backward: Assume that \([M] \longrightarrow \delta \, a\). Since reduction is deterministic in the untyped calculus \(a\) must be \([V_i]\). Hence, we may take \(V_i\) for \(M'\).
Proof of Lemma 25

Assume \( \Gamma \vdash M : \tau \) and \( M \rightarrow M' \). We reason by induction on the proof of reduction.

Case \( (\Lambda(\alpha) \; M_0) \; \tau \rightarrow_{\iota} M_0[\tau/\alpha] \): Observe that both \( M \) and \( M' \) erases to \( [M_0] \).

Case \( (\lambda x. M_1) \; M_2 \rightarrow_{\beta} M_1[M_2/x] \): Then \( M \) erases to \( (\lambda x. [M_1]) \; [M_1] \) which reduces to \( [M_1][[M_2]/x] \) which is the erasure of \( M' \).

Case \( \text{proj}_i \; \tau_1 \; \tau_2 \; (V_1, V_2) \rightarrow_\delta V_i \): The conclusion follows by assumption on \( \delta \)-rules.

Case \( M \) is \( E[N] \) and \( M' \) is \( E[N'] \) and \( N \rightarrow_z N' \): By induction hypothesis, we know that a certain relation (equality when \( z = \iota \) or \( 
rightarrow_z \) otherwise) holds between \([M]\) and \([N]\). By rule congruence for \( \iota \) and rule \text{Context} otherwise, the same relation holds between \( [E[M]] \) and \( [E][[N]] \), i.e. between \([M]\) and \([N]\).

Proof of Lemma 29

Case \( v \) is \( \lambda x. a_1 \): By inversion of type erasure, \( M \) is of the form \( R[\lambda x: \tau. M_1] \) where \([M_1]\) is \( a_1 \). Since \( R \) is \( \iota \)-normal, it is of the form \( \Lambda \overline{\alpha}.[] \overline{\tau} \). Since \( \lambda x: \tau. M_1 \) is an arrow type, \( \overline{\tau} \) must be empty.

Case \( v \) is a partial application \( c \; v_1 \ldots v_n \): We show that then \( V \) is \( R[c \; \overline{\tau} \; V_1 \ldots V_n] \) with \([V_i] = v_i \) by induction on \( n \). If \( n \) is zero, then by inversion of type erasure, \( M \) is of the form \( R[c] \) as expected. Otherwise, by inversion of type erasure, \( M \) is an application \( R_n[M_1 M_2] \) where \([M_1]\) is the partial application \( c \; v_1 \ldots v_{n-1} \) and \([M_2]\) is \( v_n \). By induction hypothesis \( M_1 \) is \( R_1[c \; \overline{\tau} \; V_1 \ldots V_{n-1}] \) with \([V_i] = v_i \). Since \( R_1 \) is in an evaluation context, it is \( \iota \)-normal, hence of the form \( \Lambda \overline{\alpha}_1.[] \; \overline{\tau}_1 \). From the arity of \( c \), the type of \( M_1 \) is an arrow type. Thus \( \overline{\tau}_1 \) must be empty. Since \( R \) is applied to \( M_2 \) it cannot be a type abstraction either. Hence, \( R_1 \) is empty. Moreover, by induction hypothesis \( M_2 \) is a value \( V_n \). Hence \( M \) is \( R_n[c \; \overline{\tau} \; V_1 \ldots V_{n-1} \; V_n] \), as expected.

Proof of Corollary 30

By Lemma 28, \( M \) is of the form \( R[M_0 M_2] \) where \([M_0]\) is a value \( v \), which is either \( \lambda x. a_1 \) or the partial application \( c \; v_1 \ldots v_{n-1} \) and \([M_2]\) is \( v \). Since \( R \) is an evaluation context, \( M_0 \; M_2 \) is in \( \iota \)-normal form. Since \([\; ]\) \( M_2 \) is an evaluation context, \( M_0 \) is in \( \iota \)-normal form. By Lemma 29, \( M_0 \) a value \( V_0 \). Since \( V_0 \) is an evaluation context, \( M_2 \) is in \( \iota \)-normal form. By 29 it must be a value \( V_0 \).

Moreover, by Lemma 29, \( V_0 \) is either

- \( \Lambda \overline{\alpha}. \lambda x: \tau. M_1 \). Since \( V_0 \) is in application position \( \overline{\alpha} \) must actually be empty. Then \( M \) is of the form \( R[(\lambda x: \tau. M_1) \; V] \), as expected.
\[ R_0[c \overline{\tau} V_1 \ldots V_{n-1}] \]. Since \( V_0 \) is in an evaluation, \( R_0 \) is \( \nu \)-normal, thus of the form \( \Lambda \overline{\alpha} . [\] \). Since \( V_0 \) in application position \( \overline{\alpha} \) must be empty. From the arity of \( d \), the application is partial and has an arrow type, hence \( \overline{\tau}_0 \) must be empty. Then, taking \( V \) for \( V_n \), the term \( M \) is of the form \( R[c \overline{\tau} V_1 \ldots V_n] \), as expected.

**Solution of Exercise 34**

Take \((\lambda x : \tau_0. \Lambda \alpha. \lambda \alpha : y . y ) \) \( M_0 \) where \( \Gamma \vdash M_0 : \tau_0 \). We have \( \Gamma \vdash M : \forall \alpha. \alpha \rightarrow \alpha \) where \( M \) is syntactically in ML, but cannot be typed in ML.
Chapter 5

Type reconstruction

5.1 Introduction

We have viewed a type system as a 3-place predicate over a type environment, a term, and a type. So far, we have been concerned with logical properties of the type system, namely subject reduction and progress. However, one should also study its algorithmic properties: is it decidable whether a term is well-typed?

We have seen three different type systems, simply-typed λ-calculus, ML, and System F, of increasing expressiveness. In each case, we have presented an explicitly-typed and an implicitly-typed version of the language and shown a close correspondence between the two views, thanks to a type-erasing semantics.

We argued that the explicitly-typed version is often more convenient for studying the meta-theoretical properties of the language. Which one should we used for checking well-typedness? That is, in which language should we write programs?

The typing judgment is inductively defined, so that, in order to prove that a particular instance holds, one exhibits a type derivation. A type derivation is essentially a version of the program where every node is annotated with a type. Checking that a type derivation is correct is usually easy: it basically amounts to checking equalities between types. However, type derivations are too verbose to be tractable by humans! Requiring every node to be type-annotated is not practical.

A more practical, common approach consists in requesting just enough annotations to allow types to be reconstructed in a bottom-up manner. In other words, one seeks an algorithmic reading of the typing rules, where, in a judgment $\Gamma \vdash M : \tau$, the parameters $\Gamma$ and $M$ are inputs, while the parameter $\tau$ is an output. Moreover, typing rules should be such that a type appearing as output in a conclusion should also appear as output in a premise or as input in the conclusion; and input in the premises should be input of the conclusion or an output of other premises.

This way, types need never be guessed, just looked up into the typing context, instanti-
CHAPTER 5. TYPE RECONSTRUCTION

ated, or checked for equality. This is exactly the situation with explicitly-typed presentations of the typing rules. This is also the traditional approach of Pascal, C, C++, Java, etc.: formal procedure parameters, as well as local variables, are assigned explicit types. The types of expressions are synthesized bottom-up.

However, this implies a lot of redundancies: Parameters of all functions need to be annotated, even when their types are obvious from context; Primitive let-bindings, recursive definitions, injection into sum types need to be annotated. As the language grows, more and more constructs require type annotations, e.g. type applications and type abstractions. Type annotations may quickly obfuscate the code and large explicitly-typed terms are so verbose that they become intractable by humans! Hence, programming in the implicitly-typed version is more appealing.

For simply-typed λ-calculus and ML, it turns out that this is possible: whether a term is well-typed is decidable, even when no type annotations are provided! We first present type inference in the case of simply-typed λ-calculus taking advantage of the simplicity to introduce type constraints as a useful intermediate to mediate between the typing rules and the type-inference algorithms. We then extend type-constraint to perform type inference for ML.

For System F, type inference is undecidable. Since programming in explicitly-typed System F is not practically feasible, some amount of type reconstruction must still be done. Typically, the algorithm is incomplete, i.e. it rejects terms that are perhaps well-typed, but the user may always provide more annotations—and at least the fully annotated version is always accepted if well-typed. We will present very briefly several techniques for type reconstruction in System F.

5.2 Type inference for simply-typed λ-calculus

The type inference algorithm for simply-typed λ-calculus, is due to Hindley. The idea behind the algorithm is simple. Because simply-typed λ-calculus is a syntax-directed type system, an unannotated term determines an isomorphic candidate type derivation, where all types are unknown: they are distinct type variables. For a candidate type derivation to become an actual, valid type derivation, every type variable must be instantiated with a type, subject to certain equality constraints on types. For instance, at an application node, the type of the operator must match the domain type of the operator.

Thus, type inference for the simply-typed λ-calculus decomposes into constraint generation followed by constraint solving. Simple types are first-order terms. Thus, solving a collection of equations between simple types is first-order unification. First-order unification can be performed incrementally in quasi-linear time, and admits particularly simple solved forms.
5.2. TYPE INFERENCE FOR SIMPLY-TYPED λ-CALCULUS

$$\langle \Gamma \vdash x : \tau \rangle = \Gamma(x) = \tau$$

$$\langle \Gamma \vdash \lambda x. a : \tau \rangle = \exists \alpha_1 \alpha_2. (\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle \land \alpha_1 \rightarrow \alpha_2) \quad \text{if } \alpha_1, \alpha_2 \not\in \Gamma, \tau$$

$$\langle \Gamma \vdash a_1 a_2 : \tau \rangle = \exists \alpha. (\langle \Gamma \vdash a_1 : \alpha \rightarrow \tau \rangle \land \langle \Gamma \vdash a_2 : \alpha \rangle) \quad \text{if } \alpha \not\in \Gamma, \tau$$

Figure 5.1: constraint generation for simply-typed λ-calculus

5.2.1 Constraints

At the interface between the constraint generation and constraint solving phases is the constraint language. It is a logic: a syntax, equipped with an interpretation in a model.

There are two syntactic categories: types and constraints.

$$\begin{align*}
\tau &::= \alpha \mid F \vec{r} \\
C &::= \text{true} \mid \text{false} \mid \tau = \tau \mid C \land C \mid \exists \alpha. C
\end{align*}$$

A type is either a type variable or an arity-consistent application of a type constructor F. (The type constructors are unit, ×, +, →, etc.) An atomic constraint is truth, falsity, or an equation between types. Compound constraints are built on top of atomic constraints via conjunction and existential quantification over type variables.

Constraints are interpreted in the Herbrand universe, that is, in the set of ground types:

$$t ::= F \vec{r}$$

Ground types contain no variables. The base case in this definition is when F has arity zero. We assume that there should be at least one constructor of arity zero, so that the model is non-empty. A ground assignment \(\phi\) is a total mapping of type variables to ground types. By homomorphism, a ground assignment determines a total mapping of types to ground types.

The interpretation of constraints takes the form of a judgment, \(\phi \vdash C\), pronounced: \(\phi\) satisfies \(C\), or \(\phi\) is a solution of \(C\). This judgment is inductively defined:

$$\begin{align*}
\phi \vdash \text{true} &\quad \phi \vdash \tau_1 = \tau_2 &\quad \phi \vdash C_1 \quad \phi \vdash C_2 &\quad \phi \vdash \exists \alpha. C \\
\phi \vdash \tau_1 = \tau_2 &\quad \phi \vdash C_1 \land C_2 & &
\end{align*}$$

A constraint \(C\) is satisfiable if and only if there exists a ground assignment \(\phi\) that satisfies \(C\). We write \(C_1 \equiv C_2\) when \(C_1\) and \(C_2\) have the same solutions. The problem “given a constraint \(C\), is \(C\) satisfiable?” is first-order unification.

Type inference is reduced to constraint solving by defining a mapping \(\langle \Gamma \vdash a : \tau \rangle\) of candidate judgments to constraints, as given in Figure 5.1. Thanks to the use of existential quantification, the names that occur free in \(\langle \Gamma \vdash a : \tau \rangle\) are a subset of those that occur free in \(\Gamma\) or \(\tau\). This allows the freshness side conditions to remain local—there is no need to informally require “globally fresh” type variables.
5.2.2 A detailed example

Let us perform type inference for the closed term $\lambda f x y. (f \ x, \ f \ y)$. The problem is to construct and solve the constraint $\langle \varnothing \vdash \lambda f x y. (f \ x, \ f \ y) : \alpha_0 \rangle$, say $C$. It is possible (and, for a human, easier) to mix these tasks. A machine, however, could generate and solve in two successive phases. There are several advantages in doing this. This leads to simpler, easier to maintain code, as the generation of constraints deals with the complexity of the source language which solving may ignore; moreover, adding new construct to the language does not (in general) require new forms of constraints and can thus reuse the solving algorithm unchanged.

Solving the constraint means to find all possible ground assignments for $\alpha_0$ that satisfy the constraint. Typically, this is done by transforming the constraint into successive equivalent constraints until some constraint that is obviously satisfiable and from which solutions may be directly read.

Performing constraint generation for the 3 $\lambda$-abstractions, we have:

$$C = \exists \alpha_1 \alpha_2. \left( \exists \alpha_3 \alpha_4. \left( \exists \alpha_5 \alpha_6. \left( \langle \alpha : \alpha_1; \ x : \alpha_3; \ y : \alpha_5 \vdash (f \ x, \ f \ y) : \alpha_6 \rangle \right) \right) \right) \left( \alpha_2 = \alpha_3 \rightarrow \alpha_4 \right) \left( \alpha_0 = \alpha_1 \rightarrow \alpha_2 \right)$$

In the following, let $\Gamma$ stand for $(f : \alpha_1; \ x : \alpha_3; \ y : \alpha_5)$. We may hoist up existential quantifiers, using the rule:

$$\langle \exists \alpha. C_1 \rangle \land C_2 \equiv \exists \alpha. (C_1 \land C_2)$$

Hence, hoisting $\alpha_3$ and $\alpha_4$, and $\alpha_5$ and $\alpha_6$ twice, we get:

$$C \equiv \exists \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6. \left( \langle \Gamma \vdash (f \ x, \ f \ y) : \alpha_6 \rangle \right) \left( \alpha_4 = \alpha_5 \rightarrow \alpha_6 \land \alpha_2 = \alpha_3 \rightarrow \alpha_4 \land \alpha_0 = \alpha_1 \rightarrow \alpha_2 \right)$$

We may eliminate a type variable that has a defining equation with the rule:

$$\exists \alpha. (C \land \alpha = \tau) \equiv [\alpha \mapsto \tau]C$$

By successive elimination of $\alpha_4$ then $\alpha_2$, we get:

$$C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6. \left( \langle \Gamma \vdash (f \ x, \ f \ y) : \alpha_6 \rangle \right) \left( \alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_6 \right)$$

Let us now perform constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable ($\alpha_6$).

$$C \equiv \exists \left\{ \alpha_1 \alpha_3 \alpha_5 \atop \alpha_6 \alpha_7 \alpha_8 \right\}. \left( \langle \Gamma \vdash f \ x : \alpha_7 \rangle \right) \left( \langle \Gamma \vdash f \ y : \alpha_8 \rangle \right) \left( \alpha_7 \times \alpha_8 = \alpha_6 \right) \left( \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_6 = \alpha_0 \right) \equiv \exists \left\{ \alpha_1 \alpha_3 \alpha_5 \atop \alpha_7 \alpha_8 \right\}. \left( \langle \Gamma \vdash f \ x : \alpha_7 \rangle \right) \left( \langle \Gamma \vdash f \ y : \alpha_8 \rangle \right) \left( \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8 = \alpha_0 \right)$$
Let us focus on the first application, perform constraint generation for the variables \( f \) and \( x \) (recall that \( \Gamma \) stands for \( (f : \alpha_1; x : \alpha_3; y : \alpha_5) \)), and eliminate a type variable \( (\alpha_9) \):

\[
C_1 = \langle \Gamma \vdash f x : \alpha_7 \rangle = \exists \alpha_9. \left( \frac{\langle \Gamma \vdash f : \alpha_9 \rightarrow \alpha_7 \rangle}{\langle \Gamma \vdash x : \alpha_6 \rangle} \right) \equiv \exists \alpha_9. \left( \alpha_1 \equiv \alpha_3 \rightarrow \alpha_7 = C_2 \right)
\]

Applying this simplification under a context, with the rule:

\[
C_1 \equiv C_2 \Rightarrow \Gamma[C_1] \equiv \Gamma[C_2]
\]

we have:

\[
C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{cases} 
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \\
\langle \Gamma \vdash f y : \alpha_8 \rangle \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8 
\end{cases}
\]

We may simplify the right-hand application analogously.

\[
C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{cases} 
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \land \alpha_1 = \alpha_5 \rightarrow \alpha_8 \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8 
\end{cases}
\]

We may apply transitivity at \( \alpha_1 \), structural decomposition, and eliminate three type variables \( (\alpha_1, \alpha_5, \alpha_8) \):

\[
C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{cases} 
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \land \alpha_3 = \alpha_5 \land \alpha_7 = \alpha_8 \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8 
\end{cases}
\]

We have now reached a solved form. To sum up, we have checked the following equivalence holds:

\[
\langle \emptyset \vdash \lambda f y. (f x, f y) : \alpha_0 \rangle \equiv \exists \alpha_3 \alpha_7. \left( (\alpha_3 \rightarrow \alpha_7) \rightarrow \alpha_3 \rightarrow \alpha_3 \rightarrow \alpha_7 \times \alpha_7 = \alpha_0 \right)
\]

Hence, the ground types of \( \lambda f y. (f x, f y) \) are all ground types of the form

\[
(t_3 \rightarrow t_7) \rightarrow t_3 \rightarrow t_3 \rightarrow t_7 \times t_7
\]

In other words, \( (\alpha_3 \rightarrow \alpha_7) \rightarrow \alpha_3 \rightarrow \alpha_3 \rightarrow \alpha_7 \times \alpha_7 \) is a principal type for \( \lambda f y. (f x, f y) \).

The language OCaml implements a form of this type inference algorithm:

```ml
# fun f x y -> (f x, f y);;
- : ('a -> 'b) -> 'a -> 'a -> 'b * 'b = (fun)
```

This technique is used also by Standard ML and Haskell.

In the simply-typed \( \lambda \)-calculus, type inference works just as well for open terms. For instance, the term \( \lambda xy. (f x, f y) \) has a free variable, namely \( f \). The type inference problem is to construct and solve the constraint \( \langle f : \alpha_1 \vdash \lambda xy. (f x, f y) : \alpha_2 \rangle \). We have already done so... with only a slight difference: \( \alpha_1 \) and \( \alpha_2 \) are now free, so they cannot be eliminated.
One can check the following equivalence:
\[ \langle f : \alpha_1 \vdash \lambda xy. (f x, f y) : \alpha_2 \rangle \equiv \exists \alpha_3 \alpha_7. \left( \alpha_1 = \alpha_3 \rightarrow \alpha_7 \land \alpha_2 = \alpha_3 \rightarrow \alpha_3 \rightarrow \alpha_7 \times \alpha_7 \right) \]
In other words, the ground typings of \( \lambda xy. (f x, f y) \) are all ground typings of the form:
\[ ((f : t_3 \rightarrow t_7), t_3 \rightarrow t_3 \rightarrow t_7 \times t_7) \]
Remember that a typing is a pair of an environment and a type.

### 5.2.3 Soundness and completeness of type inference

**Definition 3 (Typing)** A pair \( (\Gamma, \tau) \) is a typing of \( a \) if and only if \( \text{dom}(\Gamma) = \text{fv}(a) \) and the judgment \( \Gamma \vdash a : \tau \) is valid.

The type inference problem is to determine whether a term \( a \) admits a typing, and, if possible, to exhibit a description of the set of all of its typings.

Up to a change of universes, the problem reduces to finding the ground typings of a term. (For every type variable, introduce a nullary type constructor. Then, ground typings in the extended universe are in one-to-one correspondence with typings in the original universe.)

**Theorem 15 (Soundness and completeness)** \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \) if and only if \( \phi \Gamma \vdash a : \phi \tau \).

In other words, assuming \( \text{dom}(\Gamma) = \text{fv}(a) \), \( \phi \) satisfies the constraint \( \langle \Gamma \vdash a : \tau \rangle \) if and only if \( (\phi \Gamma, \phi \tau) \) is a (ground) typing of \( a \). The direct implication is soundness; the reverse implication is completeness. The proof is by structural induction over \( a \). \( \Box \) (Proof p. 1231)

**Exercise 35 (Recommended)** Write the details of the proof.

**Corollary 34** Let \( \text{fv}(a) = \{x_1, \ldots, x_n\} \), where \( n \geq 0 \). Let \( \alpha_0, \ldots, \alpha_n \) be pairwise distinct type variables. Then, the ground typings of \( a \) are described by \( ((x_i : \phi \alpha_i)_{i \in 1..n}, \phi \alpha_0) \) where \( \phi \) ranges over all solutions of \( \langle (x_i : \alpha_i)_{i \in 1..n} \vdash a : \alpha_0 \rangle \).

**Corollary 35** Let \( \text{fv}(a) = \emptyset \). Then, \( a \) is well-typed if and only if \( \exists \alpha. \langle \emptyset \vdash a : \alpha \rangle \equiv \text{true} \).

### 5.2.4 Constraint solving

A constraint solving algorithm is typically presented as a (non-deterministic) system of constraint rewriting rules that must enjoy the following properties: reduction is meaning-preserving, i.e. \( C_1 \rightarrow C_2 \) implies \( C_1 \equiv C_2 \); reduction is terminating; and every normal form is either "false" (literally) or satisfiable. The normal forms are called solved forms.

Our constraints are equations on first-order terms. They can be solved by first-order unification. The algorithm can be described as constraint solving. However, in order to
5.2. TYPE INFERENCE FOR SIMPLY-TYPED $\lambda$-CALCULUS

\[(\exists \check{\alpha}. U_1) \land U_2 \rightarrow \exists \check{\alpha}. (U_1 \land U_2)\]  
(extrusion)

\[\alpha = \epsilon \land \alpha = \epsilon' \rightarrow \alpha = \epsilon = \epsilon'\]  
(fusion)

\[F \check{\alpha} = F \check{\tau} = \epsilon \rightarrow \check{\alpha} = \check{\tau} \land F \check{\alpha} = \epsilon\]  
(decomposition)

\[F \tau_1 \ldots \tau_1 \ldots \tau_n = \epsilon \rightarrow \exists \alpha. (\alpha = \tau_1 \land F \tau_1 \ldots \alpha \ldots \tau_n = \epsilon)\]  
(naming)

\[F \check{\tau} = F' \check{\tau}' = \epsilon \rightarrow false\]  
(clash)

\[U \rightarrow false\]  
(occurs check)

\[U [false] \rightarrow false\]  
(error propag.)

\[\alpha = \alpha = \epsilon \rightarrow \alpha = \epsilon\]  
(elim dupl.)

\[F \check{\tau} \rightarrow true\]  
(elim triv.)

\[U \land true \rightarrow U\]  
(elim true)

Figure 5.2: Solving unification constraints

describe an efficient algorithm, we first extend the syntax of constraints and replace ordinary binary equations with multi-equations, following Pottier and Rémy (2005, §10.6):

\[U ::= true \mid false \mid \epsilon \mid U \land U \mid \exists \check{\alpha}.U\]

A multi-equation $\epsilon$ is a multi-set of types. Its interpretation is given by

\[\forall \tau \in \epsilon, \quad \phi \tau = t\]

That is, $\phi$ satisfies $\epsilon$ if and only if $\phi$ maps all members of $\epsilon$ to a single ground type.

Simplification rules are given in Figure 5.2 (See Pottier and Rémy (2005, §10.6) for a detailed presentation.) The last three rules in gray are administrative.

The occurs check is defined as follows: we say that $\alpha$ dominates $\beta$ (with respect to $U$) if $U$ contains a multi-equation of the form $F \tau_1 \ldots \beta \ldots \tau_n = \alpha = \ldots$. A constraint $U$ is cyclic if and only if its domination relation is cyclic. A cyclic constraint is unsatisfiable: indeed, if $\phi$ satisfies $U$ and if $\alpha$ is a member of a cycle, then the ground type $\phi \alpha$ must be a strict subterm of itself, a contradiction. Thus, the occurs-check rewriting rule is meaning-preserving.

A solved form is either false or $\exists \check{\alpha}.U$, where $U$ is a conjunction of multi-equations, every multi-equation contains at most one non-variable term, no two multi-equations share a variable, and the domination relation is acyclic. Every solved form that is not false is satisfiable. Indeed, a solution is easily constructed by well-founded recursion over the domination relation.
Remarks Viewing a unification algorithm as a system of rewriting rules makes it easy to explain and reason about.

In practice, following Huet (1976), first-order unification is implemented on top of an efficient union-find data structure (Tarjan, 1975). Its time complexity is quasi-linear (i.e. growing in the inverse of the Ackermann function).

Unification on first-order terms can also be implemented in linear time, but with a more complex algorithm and a higher constant that makes it behave worse than the quasi-linear time algorithm. Moreover, while the quasi-linear time algorithm works as well when types are regular trees—by just removing the occur check—the linear time algorithm only works with finite trees and thus cannot be used for type inference in the presence of equi-recursive types.

Closing remarks Thanks to type inference, conciseness and static safety are not incompatible. Furthermore, an inferred type is sometimes more general than a programmer-intended type. Type inference helps reveal unexpected generality.

5.3 Type inference for ML

Two presentations of type inference for Damas and Milner’s type system are possible: One of Milner’s classic algorithms 1978, \( \mathcal{W} \) or \( \mathcal{J} \); see Pottier’s old course notes for details (Pottier, 2002, §3.3); or a constraint-based presentation (Pottier and Rémy, 2005). We favor the latter, but quickly review the former first.

5.3.1 Milner’s Algorithm \( \mathcal{J} \)

Milner’s Algorithm \( \mathcal{J} \) expects a pair \( \Gamma \vdash a \), produces a type \( \tau \), and uses two global variables, \( \mathcal{V} \) and \( \varphi \). Variable \( \mathcal{V} \) is an infinite fresh supply of type variables; \( \varphi \) is an idempotent substitution (of types for type variables), initially the identity. The fresh primitive is defined as:

\[
\text{fresh} = \text{do } \alpha \in \mathcal{V}; \text{ do } \mathcal{V} \leftarrow \mathcal{V} \setminus \{\alpha\}; \text{ return } \alpha
\]

The Algorithm \( \mathcal{J} \) is given on Figure 5.3 in monadic style. The algorithm mixes generation and solving of equations. This lack of modularity leads to several weaknesses; proofs are more difficult; correctness and efficiency concerns are not clearly separated (if implemented literally, the algorithm is exponential in practice); adding new language constructs duplicates solving of equations; generalizations, such as the introduction of subtyping, are not easy. Furthermore, Algorithm \( \mathcal{J} \) works with substitutions, instead of constraints. Substitutions are an approximation to solved forms for unification constraints. Working with substitutions means using most general unifiers, composition, and restriction. Working with constraints means using equations, conjunction, and existential quantification.
5.3. TYPE INFERENCE FOR ML

\[ \mathcal{J}(\Gamma \vdash x) = \begin{cases} \forall \alpha_1 \ldots \alpha_n. \tau = \Gamma(x) & \text{let } \forall \alpha_1 \ldots \alpha_n. \tau = \Gamma(x) \\ \text{do } \alpha_1', \ldots, \alpha_n' = \text{fresh}, \ldots, \text{fresh} & \\ \text{return } [\alpha_i \mapsto \alpha_i']_{i=1}^n (\tau) - \text{take a fresh instance} \end{cases} \]

\[ \mathcal{J}(\Gamma \vdash \lambda x. a_1) = \begin{cases} \text{do } \alpha = \text{fresh} & \\ \text{do } \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash a_1) & \\ \text{return } \alpha \rightarrow \tau_1 - \text{form an arrow type} \end{cases} \]

\[ \mathcal{J}(\Gamma \vdash a_1 \ a_2) = \begin{cases} \text{do } \tau_1 = \mathcal{J}(\Gamma \vdash a_1) & \\ \text{do } \tau_2 = \mathcal{J}(\Gamma \vdash a_2) & \\ \text{do } \alpha = \text{fresh} & \\ \text{do } \phi \leftarrow \text{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi & \\ \text{return } \alpha - \text{solve } \tau_1 = \tau_2 \rightarrow \alpha \end{cases} \]

\[ \mathcal{J}(\Gamma \vdash \text{let } x = a_1 \text{ in } a_2) = \begin{cases} \text{do } \tau_1 = \mathcal{J}(\Gamma \vdash a_1) & \\ \text{let } \sigma = \forall \setminus \text{ftv}(\varphi(\Gamma)). \varphi(\tau_1) - \text{generalize} & \\ \text{return } \mathcal{J}(\Gamma; x : \sigma \vdash a_2) \end{cases} \]

(\forall \setminus \bar{\alpha}. \tau \text{ quantifies over all type variables other than } \bar{\alpha}.)

Figure 5.3: Type inference algorithm for ML

5.3.2 Constraint-based type inference for ML

Type inference for Damas and Milner’s type system involves slightly more than first-order unification: there is also generalization and instantiation of type schemes. So, the constraint language must be enriched. We proceed in two steps: still within simply-typed \(\lambda\)-calculus, we present a variation of the constraint language; building on this variation, we introduce polymorphism.

How about letting the constraint solver, instead of the constraint generator, deal with environment access and construction? That is, the syntax of constraints is as follows:

\[ C ::= \ldots | x = \tau | \text{def } x : \tau \text{ in } C \]

The idea is to interpret constraints in such a way as to validate the equivalence law:

\[ \text{def } x : \tau \text{ in } C \equiv [x \mapsto \tau]C \]

The \text{def} form is an \textit{explicit substitution} form. More precisely, here is the new interpretation of constraints. As before, a valuation \(\phi\) maps type variables \(\alpha\) to ground types. In addition, a valuation \(\psi\) maps term variables \(x\) to ground types. The satisfaction judgment now takes the form \(\phi, \psi \vdash C\). The new rules of interest are:

\[ \frac{\psi x = \phi \tau}{\phi, \psi \vdash x = \tau} \quad \frac{\phi, \psi [x \mapsto \phi \tau] \vdash C}{\phi, \psi \vdash \text{def } x : \tau \text{ in } C} \]

(All other rules are modified to just transport \(\psi\).) Constraint generation becomes a mapping of an expression \(a\) and a type \(\tau\) to a constraint \(\langle a : \tau \rangle\). There is no longer a need for the
\[ \langle x : \tau \rangle = x = \tau \]
\[ \langle \lambda x. a : \tau \rangle = \exists \alpha_1 \alpha_2 . \text{(def } x : \alpha_1 \text{ in } \langle a : \alpha_2 \rangle \land \alpha_1 \rightarrow \alpha_2 = \tau \text{)} \]
if \( \alpha_1, \alpha_2 \neq a, \tau \)
\[ \langle a_1 \ a_2 : \tau \rangle = \exists \alpha . (\langle a_1 : \alpha \rightarrow \tau \rangle \land \langle a_2 : \alpha \rangle) \]
\[ \text{if } \alpha \neq a_1, a_2, \tau \]

Figure 5.4: Constraints with program variables

Theorem 16 (Soundness and completeness) Assume \( \text{fv}(a) = \text{dom}(\Gamma) \). Then, \( \phi, \phi \Gamma \vdash \langle a : \tau \rangle \) if and only if \( \phi \Gamma \vdash a : \phi \tau \).

Corollary 36 Assume \( \text{fv}(a) = \emptyset \). Then, \( a \) is well-typed if and only if \( \exists \alpha . \langle a : \alpha \rangle \equiv \text{true} \).

This variation shows that there is freedom in the design of the constraint language, and that altering this design can shift work from the constraint generator to the constraint solver, or vice-versa.

Enriching constraints To permit polymorphism, we must extend the syntax of constraints so that a variable \( x \) denotes not just a ground type, but a set of ground types.

However, these sets cannot be represented as type schemes \( \forall \alpha . \tau \), because constructing these simplified forms requires constraint solving. To avoid mingling constraint generation and constraint solving, we use type schemes that incorporate constraints, called constrained type schemes. The syntax of constraints and of constrained type schemes is:

\[
C ::= \tau = \tau \mid C \land C \mid \exists \alpha . C \mid x \leq \tau \mid \sigma \leq \tau \mid \text{def } x : \sigma \text{ in } C
\]

\[
\sigma ::= \forall \alpha [C]. \tau
\]

Both \( x \leq \tau \) and \( \sigma \leq \tau \) are instantiation constraints. The latter form is introduced so as to make the syntax stable under substitutions of constrained type schemes for variables. As before, \( \text{def } x : \sigma \text{ in } C \) is an explicit substitution form.

The idea is to interpret constraints in such a way as to validate the equivalence laws:

\[
\text{def } x : \sigma \text{ in } C \equiv [x \mapsto \sigma] C \quad (\forall \alpha [C]. \tau) \leq \tau' \equiv \exists \alpha . (C \land \tau = \tau') \quad \text{if } \bar{\alpha} \neq \tau'
\]

Using these laws, a closed constraint can be rewritten to a unification constraint (with a possibly exponential increase in size). The new constructs do not add much expressive power. They add just enough to allow a stand-alone formulation of constraint generation.

The interpretation of constraints must be redefined since the environment \( \psi \) now maps program variables to sets of ground types. The environment \( \phi \) still maps type variables to ground types. Hence, a type variable \( \alpha \) still denotes a ground type. A variable \( x \) now denotes parameter \( \Gamma \). Constraint generation is defined in Figure 5.3.
a set of ground types. Instantiation constraints are interpreted as set membership. The rules for the new form of constraints are:

\[
\begin{align*}
\phi \tau \in \psi x & \quad \phi \tau \in \varnothing \sigma \quad \phi, \psi \vdash x : \sigma \ \text{in} \ C \\
\phi, \psi \vdash x \leq \tau & \quad \phi, \psi \vdash \sigma \leq \tau \\
\phi, \psi \vdash \text{def} \ x : \sigma \ \text{in} \ C
\end{align*}
\]

The interpretation of \( \forall \alpha[C]. \tau \) under \( \phi \) and \( \psi \), written \( \left( \psi_0 \right) \left( \forall \alpha[C]. \tau \right) \) is the set of all \( \phi' \tau \), where \( \phi \) and \( \phi' \) coincide outside \( \bar{\alpha} \) and where \( \phi' \) and \( \psi \) satisfy \( C \):

\[
\left( \psi_0 \right) \left( \forall \bar{\alpha}[C]. \tau \right) \triangleq \{ \phi' \tau \mid (\phi' \setminus \bar{\alpha} = \phi' \setminus \bar{\alpha}) \land (\phi', \psi \vdash C) \}
\]

If \( C \) is empty, then \( \left( \psi_0 \right) \left( \forall \bar{\alpha}[C]. \tau \right) \) is \( \{ (\phi[\bar{\alpha} \mapsto \bar{t}]) \tau \} \). If \( \bar{\alpha} \) and \( C \) are empty, then \( \left( \psi_0 \right) \tau \) is \( \phi \tau \).

For instance, the interpretation of \( \forall \alpha[3\beta. \alpha = \beta \rightarrow \gamma]. \alpha \rightarrow \alpha \) under \( \phi \) and \( \psi \) is the set of all ground types of the form \((t \rightarrow \phi_{\gamma}) \rightarrow (t \rightarrow \phi_{\gamma})\), where \( t \) ranges over ground types. This is also the interpretation of an unconstrained typed scheme, namely \( \forall \beta. (\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \).

In fact, this is a general situation:

**Lemma 37** Every constrained type scheme is equivalent to a standard type scheme.

This result holds because constraints can be reduced to unification constraints, which have either no solution or a principal solution. This is an important property as it implies that type inference problems have principal solutions and typable programs have principal types. The property would not hold with more general constraints, such as subtyping constraints. However, we may then generalize type schemes to constrained type schemes as a way to factor several possible types and recover principality of type inference. Then, type inference may have principal constrained type schemes.

Notice that if \( x \) does not appear free in \( C \), \( \text{def} \ x : \sigma \ \text{in} \ C \) is equivalent to \( C \)—whether or not the constraints appearing in \( \sigma \) are solvable. To enforce the constraints in \( \sigma \) to be solvable, we use a variant of the \( \text{def} \) construct:

\[
\text{let} \ x : \sigma \ \text{in} \ C \triangleq \ \text{def} \ x : \sigma \ \text{in} \ ((\exists \alpha. x \leq \alpha) \land C')
\]

Expanding the constraint type scheme \( \sigma \) of the form \( \forall \alpha[C]. \tau \) and simplifying, an equivalent definition is:

\[
\text{let} \ x : \forall \alpha[C]. \tau \ \text{in} \ C' \triangleq \exists \bar{\alpha}. C \land \text{def} \ x : \forall \bar{\alpha}[C]. \tau \ \text{in} \ C''
\]

This is equivalent to providing a direct interpretation of let-bindings as:

\[
\left( \psi_0 \right) \sigma \notin \varnothing \quad \phi, \psi \vdash x \mapsto \left( \psi_0 \right) \sigma \vdash C
\]

\[
\phi, \psi \vdash \text{let} \ x : \sigma \ \text{in} \ C
\]

Constraint generation for ML is defined in Figure 5.5. The abbreviation \( (a) \) is a principal constrained type scheme for \( a \): its intended interpretation is the set of all ground types that \( a \) admits.
CHAPTER 5. TYPE RECONSTRUCTION

\[
\begin{align*}
\langle x : \tau \rangle & = x \leq \tau \\
\langle \lambda x. a : \tau \rangle & = \exists \alpha_1 \alpha_2. (\text{def } x : \alpha_1 \in \langle a : \alpha_2 \rangle \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \quad \text{if } \alpha_1, \alpha_2 \neq a, \tau \\
\langle a_1 a_2 : \tau \rangle & = \exists \alpha. (\langle a_1 : \alpha \rightarrow \tau \rangle \wedge \langle a_2 : \alpha \rangle) \\
\langle \text{let } x = a_1 \text{ in } a_2 : \tau \rangle & = \text{let } x : \langle a_1 \rangle \text{ in } \langle a_2 : \tau \rangle \\
\langle a \rangle & = \forall \alpha [\langle a : \alpha \rangle] \alpha
\end{align*}
\]

Figure 5.5: Constraint generation for ML

Lemma 38 (Constraint equivalences) The following equivalences hold:

(1) \(\exists \alpha. (\langle a : \alpha \rangle \wedge \alpha = \tau) \equiv \langle a : \tau \rangle\) if \(\alpha \neq \tau\)

(2) \(\langle a \rangle \leq \tau \equiv \langle a : \tau \rangle\)

(3) \([x \mapsto \langle a_1 \rangle] \langle a_2 : \tau \rangle \equiv \langle x \mapsto a_1 \rangle a_2 : \tau\)

Proof: (1) is by induction on the definition of \(\langle a : \tau \rangle\); (2) is by definition of \(\langle a \rangle\), expansion of the instantiation constraint and (1); (3) is by induction on \(\langle a : \tau \rangle\) and (2).

Another key property is that the constraint associated with a let construct is equivalent to the constraint associated with its let-normal form.

Lemma 39 (let expansion) \(\langle \text{let } x = a_1 \text{ in } a_2 : \tau \rangle \equiv \langle a_1; [x \mapsto a_1] a_2 : \tau \rangle\).

Expansion of let-binding terminates, since it can be seen as reducing the family of redexes marked as let-bindings. The resulting expression has no let-binding and its constraint has no def-constraint. Hence, its interpretation is the same as constraints for the simply-typed \(\lambda\)-calculus. This gives another specification of ML: a closed program is well-typed in ML if and only if its let-expansion is typable with simple types.

Constraint generation for ML can still be implemented in linear time and space.

Lemma 40 The size of \(\langle a : \tau \rangle\) is linear in the sum of the sizes of \(a\) and \(\tau\).

The statement of soundness and completeness keeps its previous form, but \(\Gamma\) now contains Damas-Milner type schemes. Since \(\Gamma\) binds variables to type schemes, we define \(\phi(\Gamma)\) as the point-wise mapping of \(\langle a \rangle\) to \(\Gamma\).

Theorem 17 (Soundness and completeness) Assume \(\text{fv}(a) = \text{dom}(\Gamma)\). Then, \(\phi, \phi \Gamma \vdash \langle a : \tau \rangle\) if and only if \(\phi \Gamma \vdash a : \phi \tau\).
Key points Notice that constraint generation has linear complexity; constraint generation and constraint solving are separate. This makes constraints suitable for use in an efficient and modular implementation. In particular, the constraint language remains small as the programming language grows.

5.3.3 Constraint solving by example

For our running example, assume that the initial environment $\Gamma_0$ stands for $\text{assoc} : \forall \alpha \beta. \alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta$. That is, the constraints considered next are implicitly wrapped within the context $\text{def} \; \Gamma_0$ in []. Let $a$ stand for the term:

$$\lambda x. \lambda l_1. \lambda l_2. \text{let assoc} \equiv \text{assoc } x \in (\text{assoc} \; l_1, \text{assoc} \; l_2)$$

One may anticipate that $\text{assoc}$ receives a polymorphic type scheme, which is instantiated twice at different types. Let $\Gamma$ stand for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$. Then, the constraint $\{a : \alpha\}$ is, after a few minor simplifications:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta. \left( \begin{array}{c} \alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \text{def} \; \Gamma \; \text{in} \\ \exists \gamma_1: \forall \gamma_1 \left[ \exists \gamma_2. \left( \begin{array}{c} \text{assoc} \leq \gamma_2 \rightarrow \gamma_1 \\ x \leq \gamma_2 \end{array} \right) \right], \gamma_1 \; \text{in} \\ \exists \beta_1 \beta_2. \left( \begin{array}{c} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1, 2\}, \exists \gamma_2. \left( \text{assoc} \leq \gamma_2 \rightarrow \beta_i \land l_i \leq \gamma_2 \right) \end{array} \right) \end{array} \right)$$

Constraint solving can be viewed as a rewriting process that exploits equivalence laws. Because equivalence is, by construction, a congruence, rewriting is permitted within an arbitrary context. For instance, environment access is allowed by the law

$$\text{let } x : \sigma \text{ in } \mathcal{R}[x \leq \tau] \quad \equiv \quad \text{let } x : \sigma \text{ in } \mathcal{R}[\sigma \leq \tau]$$

where $\mathcal{R}$ is a context that does not bind $x$. Thus, within the context $\text{def} \; \Gamma_0; \Gamma$ in [], we have the following equivalence:

$$\text{assoc} \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2 \quad \equiv \quad \exists \alpha \beta. (\alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_2 \rightarrow \gamma_1) \land \alpha_0 = \gamma_2$$

By first-order unification, we have the following sequence of simplifications:

$$\exists \gamma_2. (\exists \alpha \beta. (\alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_2 \rightarrow \gamma_1) \land \alpha_0 = \gamma_2)$$

$$\equiv \exists \gamma_2. (\exists \alpha \beta. (\alpha = \gamma_2 \land \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_1) \land \alpha_0 = \gamma_2)$$

$$\equiv \exists \gamma_2. (\exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1) \land \alpha_0 = \gamma_2)$$

$$\equiv \exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1)$$

Hence,

$$\forall \gamma_1 [\exists \gamma_2. (\text{assoc} \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2)]. \gamma_1 \quad \equiv \quad \forall \gamma_1 [\exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1)]. \gamma_1$$

$$\equiv \forall \gamma_1 [\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1]. \gamma_1$$

$$\equiv \forall \beta. \text{list} (\alpha_0 \times \beta) \rightarrow \beta$$
We have used the rule:

\[ \forall \alpha [ \exists \beta. C], \tau \equiv \forall \alpha \beta [C], \tau \quad \text{if } \beta \neq \tau \]

The initial constraint has now been simplified down to:

\[
\exists \alpha_0 \alpha_1 \alpha_2 \beta.
\begin{aligned}
\alpha = \alpha_0 & \to \alpha_1 & \to \alpha_2 & \to \beta \\
def \Gamma & \text{ in } \text{let } assocx : \forall \beta. \text{list} (\alpha_0 \times \beta) & \to \beta & \text{ in } \\
\exists \beta_1 \beta_2. & (\beta = \beta_1 \times \beta_2 \\
& \forall i \in \{1, 2\}, \exists \gamma_2. (assocx \leq \gamma_2 \to \beta_i \land l_i \leq \gamma_2) \\
\end{aligned}
\]

The simplification work spent on assocx's type scheme was well worth the trouble, because we are now going to duplicate the simplified type scheme.

The subconstraint \( \exists \gamma_2. (assocx \leq \gamma_2 \to \beta_i \land l_i \leq \gamma_2) \) where \( i \in \{1, 2\} \), is rewritten:

\[
\exists \gamma_2. (\exists \beta. (\text{list} (\alpha_0 \times \beta) \to \beta = \gamma_2 \to \beta_i) \land \alpha_i = \gamma_2)
\equiv
\exists \beta. (\text{list} (\alpha_0 \times \beta) \to \beta = \alpha_i \to \beta_i)

\equiv
\exists \beta. (\text{list} (\alpha_0 \times \beta) = \alpha_i \land \beta = \beta_i)

\equiv
\text{list} (\alpha_0 \times \beta_i) = \alpha_i
\]

The initial constraint has now been simplified down to:

\[
\exists \alpha_0 \alpha_1 \alpha_2 \beta.
\begin{aligned}
\alpha = \alpha_0 & \to \alpha_1 & \to \alpha_2 & \to \beta \\
def \Gamma & \text{ in } \text{let } assocx : \forall \beta. \text{list} (\alpha_0 \times \beta) & \to \beta & \text{ in } \exists \beta_1 \beta_2. \\
(\beta = \beta_1 \times \beta_2 \\
& \forall i \in \{1, 2\}, \text{list} (\alpha_0 \times \beta_i) = \alpha_i) \\
\end{aligned}
\]

Now, the context def \( \Gamma \) in let assocx: ... in \([\ ]\) can be dropped, because the constraint that it applies to contains no occurrences of \( x, l_1, l_2, \) or assocx. The constraint becomes:

\[
\exists \alpha_0 \alpha_1 \alpha_2 \beta.
\begin{aligned}
\alpha = \alpha_0 & \to \alpha_1 & \to \alpha_2 & \to \beta \\
\exists \beta_1 \beta_2. & (\beta = \beta_1 \times \beta_2 \\
& \forall i \in \{1, 2\}, \text{list} (\alpha_0 \times \beta_i) = \alpha_i) \\
\end{aligned}
\]

that is, by extrusion:

\[
\exists \alpha_0 \alpha_1 \alpha_2 \beta_1 \beta_2.
\begin{aligned}
\alpha = \alpha_0 & \to \alpha_1 & \to \alpha_2 & \to \beta \\
\beta = \beta_1 \times \beta_2 \\
& \forall i \in \{1, 2\}, \text{list} (\alpha_0 \times \beta_i) = \alpha_i \\
\end{aligned}
\]

Finally, by eliminating a few auxiliary variables:

\[
\exists \alpha_0 \beta_1 \beta_2. (\alpha = \alpha_0 \to \text{list} (\alpha_0 \times \beta_1) \to \text{list} (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2)
\]

We have shown the following equivalence between constraints:

\[ \text{def } \Gamma_0 \text{ in } \langle a : \alpha \rangle \equiv \exists \alpha_0 \beta_1 \beta_2. (\alpha = \alpha_0 \to \text{list} (\alpha_0 \times \beta_1) \to \text{list} (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2) \]

That is, the principal type scheme of \( a \) relative to \( \Gamma_0 \) is

\[ \langle a \rangle_\alpha = \forall \alpha [\langle a : \alpha \rangle_\alpha], \alpha \equiv \forall \alpha_0 \beta_1 \beta_2. \alpha_0 \to \text{list} (\alpha_0 \times \beta_1) \to \text{list} (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2 \]
5.3. TYPE INFERENCE FOR ML

Again, constraint solving can be explained in terms of a small-step rewrite system. Again, one checks that every step is meaning-preserving, that the system is normalizing, and that every normal form is either literally “false” or satisfiable.

Rewriting strategies Different constraint solving strategies lead to different behaviors in terms of complexity, error explanation, etc. See Pottier and Rémy (2005) for details on constraint solving. See Jones (1999b) for a different presentation of type inference, in the context of Haskell.

In all reasonable strategies, the left-hand side of a let constraint is simplified before the let form is expanded away. This corresponds, in Algorithm $J$, to computing a principal type scheme before examining the right-hand side of a let construct.

Complexity Type inference for ML is DEXPTIME-complete (Kfoury et al., 1990; Mairson, 1990), so any constraint solver has exponential complexity. This is assuming that types are printed as trees. If one allows to return types are dags graphs instead of types, the complexity is EXPTIME-complete.

This is, of course, worse case complexity, which does not contradict the observation that ML type inference works well in practice.

If fact, this good behavior can be explain by the results of McAllester (2003): under the hypotheses that types have bounded size and let forms have bounded left-nesting depth, constraints can be solved in linear time, or in quasi-linear time if recursive types are allowed.

When the size of types in unbounded, one may reach worst case complexity but right-nesting let-bindings as in Mairson original example:

```ml
let mairson =
  let f = fun x → (x, x) in
  (* ... n times ... *)
  let f = fun x → f(f x) in
  f (fun z → z)
```

This term can be placed in the context `let x = ... in ()` to ignore the time spent outputing the result type.

However, this right-nesting of let-bindings is not a problem if types remain bounded, because each let-bound expression can be simplified to a type of bounded size before being duplicated.

On the opposite, in a left-nesting of let-binding local variables may have to be extruded step by step from the inner bindings to its enclosing binding, sometimes all the way up to the root, leading to a quadratic complexity when the nesting is proportional to the size of the program.

Principal constraint type schemes In constraint generation, we introduced principal constraint type scheme `⟨a⟩` as an abbreviation for `∀α[⟨a : α⟩], α`. However, using the equiv-
\[ \langle x \rangle = \forall \alpha [x \leq \alpha]. \alpha \]
\[ \langle \lambda x.a \rangle = \forall \alpha_1 \alpha_2 [\text{def } x : \alpha_2 \in \langle a \rangle \leq \alpha_1]. \alpha_2 \rightarrow \alpha_1 \]
\[ \text{if } \alpha_1, \alpha_2 \neq a \]
\[ \langle a_1 a_2 \rangle = \forall \alpha_1 \alpha_2 [\langle a_1 \rangle \leq \alpha_2 \rightarrow \alpha_1 \wedge \langle a_2 \rangle \leq \alpha_2]. \alpha_1 \]
\[ \text{if } \alpha_1, \alpha_2 \neq a, a_2 \]
\[ \langle \text{let } x = a_1 \text{ in } a_2 \rangle = \forall \alpha [\text{let } x : \langle a_1 \rangle \text{ in } \langle a_2 \rangle \leq \alpha]. \alpha \]

Figure 5.6: Constraint generation with principal constraint type schemes

alence between \( \langle a : \tau \rangle \) and \( \langle a \rangle \leq \tau \), we may conversely use principal constraint type schemes in place of program constraints. This leads to an alternative presentation of constraint generation described in Figure 5.6 (Compare it with the previous definition in Figure 5.5).

5.3.4 Type reconstruction

Type inference should not just return a principal type for an expression; it should also perform type reconstruction, i.e. elaborate the implicitly-typed input term into an explicitly-typed one.

The elaborated term is not unique, since redundant type abstractions and type applications may always be used. Moreover, some non principal type schemes may also be used for local let-bindings—even if the final type is principal.

For example the implicitly-typed term \( \text{let } x = \lambda y.y \text{ in } x \ 1 \) may be explicitly typed as either one of

\[ \text{let } x : \text{int} \rightarrow \text{int} = \lambda y: \text{int}. y \text{ in } x \ 1 \]
\[ \text{let } x : \forall \alpha. \alpha \rightarrow \alpha = \Lambda x. \lambda x: \text{int}. x \text{ in } x \text{ int} \ 1 \]

Which one is better? Monomorphic terms can be compiled more efficiently, so removing useless polymorphism may be useful.

However, one usually infers more general explicitly-typed terms. Given explicitly-typed terms \( M \) and \( M' \) with the same type erasure, we say that \( M \) is more general than \( M' \) if all let-bindings are assigned more general type schemes in \( M \) than in \( M' \), i.e.:

for all decompositions of \( M \) into \( C[\text{let } x : \sigma = M_1 \text{ in } M_2] \), then there is a corresponding decomposition of \( M' \) (i.e. one where \( C \) and \( C' \) have the same erasure) as \( C'[\text{let } x : \sigma' = M'_1 \text{ in } M'_2] \) where \( \sigma \) is more general than \( \sigma' \).

A type reconstruction is principal if it is more general than any other type reconstruction of the same term. Core ML admits principal type reconstructions. A principal typing derivation can be sought for in canonical form, as defined in 4.6.2.

A term in canonical form is uniquely determined up to reordering of type abstractions and type applications by the type schemes of bound program variables and of how they are
5.3. TYPE INFERENCE FOR ML

instance. We may keep track of such information during constraint resolution by keeping the binding constraints \( \text{def } x : C \in C \) and its derived form \( \text{let } x : C \in C \), and the instantiation constraints \( x \leq \tau \) of the original constraint—instead of removing them once solved. We call them persistent constraints. We thus forbid the removal, as well as the extrusion of persistent constraints by restricting the equivalence of constraints accordingly.

Rewriting rules used for constraint resolution can easily be adapted to retain the persistent constraints—and thus preserve the restricted notion of equivalence. Then, the binding structure of the constraint remains unchanged during simplification and is isomorphic to the binding structure of the expression it came from. (Persistent nodes could actually be labeled by their corresponding nodes in the original expression.)

In practice, we mark nodes of the persistent constraints as resolved when they could have been dropped in the normal resolution process—so that they need not be considered anymore during the resolution. For example, we use the rule

\[
\text{def } x : \sigma \in R[x \leq \tau] \equiv \text{def } x : \sigma \in R[x \leq \tau \wedge \sigma \leq \tau]
\]

for environment access, where the original constraint \( x \leq \tau \) is kept and marked as resolved but is not removed. Similarly, a constraint \( \text{def } x : \sigma \in C \) can be marked as resolved, which we write \( \text{def } x : \sigma \in C \), whenever \( x \) may only appears free in removable constraints of \( C \). A resolved form of a constraint is an equivalent persistent constraint, such that dropping all persistent nodes is an equivalent constraint in solved forms.

For example, reusing the running example and notations of the previous section, let us find a term \( M \) whose erasure \( a \) is defined as:

\[
\lambda x. \lambda l_1. \lambda l_2. \text{let associ } = \text{assoc } x \in (\text{assoc } l_1, \text{assoc } l_2)
\]

The principal type scheme \( \llparenthesis a \rrparenthesis \) is, by definition:

\[
\forall \alpha \exists \alpha_0 \alpha_1 \alpha_2 \beta. \left( \begin{array}{c}
\text{def } \Gamma \text{ in } \\
\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\
\text{let associ } : \forall \gamma_1 \exists \gamma_2. \left( \begin{array}{c}
\text{associ } \leq \gamma_2 \rightarrow \gamma_1 \\
x \leq \gamma_2
\end{array} \right), \gamma_1 \text{ in } \\
\exists \beta_1 \beta_2. \left( \begin{array}{c}
\beta = \beta_1 \times \beta_2 \\
\forall i \in \{1, 2\}, \exists \gamma_i. (\text{associ } \leq \gamma_i \rightarrow \beta_i \land l_i \leq \gamma_2)
\end{array} \right) \] \right)
\]

Since \( x : \alpha_9 \) is in \( \Gamma \), the inner constraint can be resolved as follows:

\[
\exists \gamma_2. (\text{associ } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2)
\equiv
\exists \gamma_2. (\text{associ } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2 \land \alpha_0 \leq \gamma_2) \equiv \text{associ } \leq \alpha_0 \rightarrow \gamma_1 \land x \leq \alpha_0
\]

The other instantiation may be solved similarly, leading to the equivalent constraints:

\[
\text{associ } \leq \alpha_0 \rightarrow \gamma_1 \land \forall \alpha \beta. \alpha \rightarrow \text{list } (\alpha \times \beta) \rightarrow \beta \leq \alpha_0 \rightarrow \gamma_1 \land x \leq \alpha_0
\equiv
\text{associ } \leq \alpha_0 \rightarrow \gamma_1 \land \exists \alpha \beta. (\alpha = \alpha_0 \land \text{list } (\alpha \times \beta) \rightarrow \beta = \gamma_1) \land x \leq \alpha_0
\equiv
\exists \beta. (\text{associ } \leq \alpha_0 \rightarrow \text{list } (\alpha_0 \times \beta) \rightarrow \beta \land \text{list } (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1 \land x \leq \alpha_0)
\]
Hence, the type scheme of assoc is equivalent to

$$\forall \beta [\text{assoc} \leq \alpha_0 \to \text{list} (\alpha_0 \times \beta) \to \beta \times x \leq \alpha_0]. \text{list} (\alpha_0 \times \beta) \to \beta$$

and \(\{a_1\}\) is equivalent to:

$$\forall \alpha \left[ \exists \alpha_0 \alpha_1 \alpha_2 \beta. \left( \begin{array}{l}
\alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta
\\
def \Gamma \text{ in }
\\let \text{assoc} : \forall \beta [\text{assoc} \leq \alpha_0 \to \text{list} (\alpha_0 \times \beta) \to \beta \times x \leq \alpha_0]. \text{list} (\alpha_0 \times \beta) \to \beta \text{ in }
\\and \exists \beta_1 \beta_2. \left( \begin{array}{l}
\beta = \beta_1 \times \beta_2
\\\forall i \in \{1, 2\}, \exists \gamma_i. (\text{assoc} \leq \gamma_i \to \beta_i \times l_i \leq \gamma_i)
\end{array} \right)
\end{array} \right) \right]. \alpha$$

Simplifying the remaining instantiation constraints in a similar way, we end up with the following resolved type scheme for \(\{a\}\):

$$\forall \alpha_0 \beta_1 \beta_2 \left[ \begin{array}{l}
def \Gamma \text{ in }
\\let \text{assoc} : \forall \gamma [\text{assoc} \leq \alpha_0 \to \text{list} (\alpha_0 \times \gamma) \to \gamma]. \text{list} (\alpha_0 \times \gamma) \to \gamma \text{ in }
\\\forall i \in \{1, 2\}, \exists \gamma_i. (\text{assoc} \leq \gamma_i \to \beta_i \times l_i \leq \gamma_i)
\\\alpha_0 \to \text{list} (\alpha_0 \times \beta_1) \to \text{list} (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2
\end{array} \right]. \alpha$$

This is a resolved form, from which we may build the elaboration of \(a_1\):

$$\Lambda \alpha_0 \beta_1 \beta_2. \lambda x : \alpha_0. \lambda l_1 : \text{list} (\alpha_0 \times \beta_1). \lambda l_2 : \text{list} (\alpha_0 \times \beta_2).\ 
\text{let assoc} = \Lambda \gamma. \text{assoc} \alpha_0 \gamma \ x \in (\text{assoc} \beta_1 \ l_1, \text{assoc} \beta_2 \ l_2)$$

Type abstractions are determined by their corresponding type scheme in the resolved constraint; for instance, the type abstraction for the let-bound variable \text{assoc} is \(\gamma\) while the toplevel type abstraction is \(\alpha_0 \alpha_1 \beta_2\). Type annotations on abstractions are determined by \(\Gamma\), which here contains \(x : \alpha_0; l_1 : \text{list} (\alpha_0 \times \alpha_1); l_2 : \text{list} (\alpha_0 \times \alpha_2)\). Type applications are inferred locally by looking at their corresponding type instantiations in the resolved constraints. For instance, we read from the constraint that \text{assoc} is let-bound with the type scheme \(\forall \gamma. \text{list} (\alpha_0 \times \gamma) \to \gamma\) (we dropped the constraint which is solved and equivalent to \text{true}) and that its \(i\)-th occurrence is used at type \(\text{list} (\alpha_0 \times \beta_i) \to \beta_i\). Matching the former against the latter gives the substitution \(\gamma \mapsto \beta_i\). Therefore, the type application for the \(i\)'s occurrence is \text{be} \(\beta_i\).

**Principal type reconstruction** Notice that while the constraint framework enforces the inference of principal types, since it transforms the original constraint into an equivalent constraint, it does not enforce type reconstruction to be principal. Indeed, in a constraint \(\exists \alpha. C\), the existentially bound type variable \(\alpha\) may be instantiated to any type that satisfies the constraint \(C\) and not necessarily the most general one.

Interestingly, however, the default strategy for constraint resolution always returns principal type reconstructions. That is, variables are never arbitrarily instantiated, although this
would be allowed by the specification.

Exercise 36 (Minimal derivations) On the opposite, one may seek for less general typing derivations where all let-expressions are as instantiated as possible. Do such derivations exists? In fact no: there are examples where there are two minimal incomparable type reconstructions and others with smaller and smaller type reconstructions but no smallest one. Find examples of both kinds. (Solution p. 125)

Exercise 37 (Closed types) Explain why ML modules in combination with the value-restriction break the principal type property: that is, there are programs that are typable but that do not have a principal type. Hint: ML signatures of ML modules must be closed. (Solution p. 125)

5.4 Type annotations

Damas and Milner’s type system has principal types: at least in the core language, no type information is required. This is very lightweight, but a bit extreme: sometimes, it is useful to write types down, and use them as machine-checked documentation. Let us, then, allow programmers to annotate a term with a type:

\[ a :: \tau \]

Typing and constraint generation are obvious:

\[
\begin{align*}
\Gamma \vdash a : \tau \\
\Gamma \vdash (a : \tau) : \tau
\end{align*}
\]

\[
\langle (a : \tau) : \tau' \rangle = \langle a : \tau \rangle \land \tau = \tau'
\]

Type annotations are erased prior to runtime, so the operational semantics is not affected. In particular, it is still type-erasing.

Notice that annotations here do not help type more terms, as erasure of type annotations preserves well-typedness: Indeed, the constraint \(\langle (a : \tau) : \tau' \rangle\) implies the constraint \(\langle a : \tau' \rangle\). That is, in terms of type inference, type annotations are restrictive: they lead to a principal type that is less general, and possibly even to ill-typedness. For instance, \(\lambda x. x\) has principal type scheme \(\forall \alpha. \alpha \rightarrow \alpha\), whereas \((\lambda x. x : \text{int} \rightarrow \text{int})\) has principal type scheme \(\text{int} \rightarrow \text{int}\), and \((\lambda x. x : \text{int} \rightarrow \text{bool})\) is ill-typed.

5.4.1 Explicit binding of type variables

We must be careful with type variables within type annotations, as in, say:

\[(\lambda x. x : \alpha \rightarrow \alpha) \quad (\lambda x. x + 1 : \alpha \rightarrow \alpha) \quad \text{let } f = (\lambda x. x : \alpha \rightarrow \alpha) \text{ in } (f 0, f \text{ true})\]
Does it make sense, and is so, what does it mean? A short answer is that it does not mean anything, because $\alpha$ is unbound. “There is no such thing as a free variable” (Alan Perlis). A longer answer is that it is necessary to specify how and where variables are bound.

How is $\alpha$ bound? If $\alpha$ is existentially bound, or flexible, then both $(\lambda x. x : \alpha \to \alpha)$ and $(\lambda x. x + 1 : \alpha \to \alpha)$ should be well-typed. If it is universally bound, or rigid, only the former should be well-typed.

Where is $\alpha$ bound? If $\alpha$ is bound within the left-hand side of this “let” construct, then let $f = (\lambda x : \alpha \to \alpha)$ in ($f 0$, $f$ true) should be well-typed. On the other hand, if $\alpha$ is bound outside this “let” form, then this code should be ill-typed, since no single ground value of $\alpha$ is suitable.

Programmers should explicitly bind type variables. We extend the syntax of expressions as follows:

$\alpha : : = \ldots \mid \exists \bar{\alpha}. a \mid \forall \bar{\alpha}. a$

It now makes sense for a type annotation $(a : \tau)$ to contain free type variables—as long as these type variables have been introduced in some enclosing term.

Since terms can now contain free type variables, some side conditions have to be updated (e.g., $\bar{\alpha} \# \Gamma, a$ in $\text{GEN}$). The new (and updated) typing rules are as follows:

$$
\begin{align*}
\text{exists} & : \quad \Gamma \vdash [\bar{\alpha} \mapsto \bar{\tau}]a : \tau \\
\text{forall} & : \quad \Gamma \vdash a : \tau \quad \bar{\alpha} \# \Gamma \\
\text{gen} & : \quad \Gamma \vdash a : \forall \bar{\alpha}. \tau
\end{align*}
$$

As type annotations, the introduction of type variables are erased prior to runtime.

**Exercise 38** Define the erasure of implicitly-typed terms and show that the erasure of a well-typed term is well-typed. Use this to justify the soundness of the extension of ML with type annotations with explicit introduction of type variables.

Constraint generation for the existential form is straightforward:

$$\langle (\exists \bar{\alpha}. a) : \tau \rangle = \exists \bar{\alpha}. \langle a : \tau \rangle$$

if $\bar{\alpha} \# \tau$

The type annotations inside $a$ contain free occurrences of $\bar{\alpha}$. Thus, the constraint $\langle a : \tau \rangle$ contains such occurrences as well, which are bound by the existential quantifier.

For example, the expression $\lambda x_1. \lambda x_2. \exists \alpha.((x_1 : \alpha), (x_2 : \alpha))$ has principal type scheme $\forall \alpha. \alpha \to \alpha \to \alpha \times \alpha$. Indeed, the generated constraint is of the form $\exists \alpha.\langle \langle x_1 : \alpha \rangle \land \langle x_2 : \alpha \rangle \land \ldots \rangle$, which requires $x_1$ and $x_2$ to share a common (unspecified) type.

Perhaps surprisingly, constraint generation for the universal case is more difficult. A term $a$ has type scheme, say, $\forall \alpha. \alpha \to \alpha$ if and only if $a$ has type $\alpha \to \alpha$ for every instance of $\alpha$, or, equivalently, for an abstract $\alpha$. To express this in terms of constraints, we introduce universal quantification in the constraint language:

$$C := \ldots \mid \forall \alpha. C$$
Its interpretation is as expected:

\[
\forall t, \phi[\alpha \mapsto t], \psi \vdash C \\
\phi, \psi \vdash \forall \alpha. C
\]

(To solve these constraints, we will use an extension of the unification algorithm called unification under a mixed prefix—see §5.4.3.)

The need for universal quantification in constraints arises when polymorphism is required by the programmer, as opposed to inferred by the system. Constraint generation for the universal form is somewhat subtle. A naive definition fails:

\[
\llangle \forall \bar{\alpha}. a : \tau \rrangle = \forall \bar{\alpha}. \llangle a : \tau \rrangle \quad \text{if } \bar{\alpha} \neq \tau
\]

Wrong!

This requires \( \tau \) to be simultaneously equal to all of the types that \( a \) assumes when \( \bar{\alpha} \) varies. For instance, with this incorrect definition, one would have:

\[
\llangle \forall \alpha. (\lambda x. x : \alpha \to \alpha) : \text{int} \to \text{int} \rrangle
\]

\[
= \forall \alpha. \llangle (\lambda x. x : \alpha \to \alpha) : \text{int} \to \text{int} \rrangle
\]

\[
= \forall \alpha. (\llangle \lambda x. x : \alpha \to \alpha \rrangle \land \alpha = \text{int})
\]

\[
\equiv \forall \alpha. (\text{true} \land \alpha = \text{int})
\]

\[
\equiv \text{false}
\]

A correct definition is:

\[
\llangle \forall \bar{\alpha}. a : \tau \rrangle = \forall \bar{\alpha}. \exists \gamma. \llangle a : \gamma \rrangle \land \exists \bar{\alpha}. \llangle a : \tau \rrangle
\]

This requires \( a \) to be well-typed for all instances of \( \bar{\alpha} \) and requires \( \tau \) to be a valid type for \( a \) under some instance of \( \bar{\alpha} \).

However, a problem with this definition is that the term \( a \) is duplicated, which can lead to exponential complexity. Fortunately, this can be avoided modulo a slight extension of the constraint language (Pottier and Rémy, 2003, p. 112). The solution defines:

\[
\llangle \forall \bar{\alpha}. a : \tau \rrangle = \text{let } x : \forall \bar{\alpha}, \beta [\llangle a : \beta \rrangle], \beta \text{ in } x \leq \tau
\]

where the new constrain form satisfies the equivalence:

\[
\text{let } x : \forall \bar{\alpha}, \beta [C_1], \tau \text{ in } C2 \equiv \forall \bar{\alpha}. \exists \bar{\beta}. C_1 \land \text{def } x : \forall \bar{\alpha}, \beta [C_1], \tau \text{ in } C2
\]

Annotating a term with a type scheme, rather than just a type, is now just syntactic sugar:

\[
(a : \forall \bar{\alpha}. \tau) \triangleq \forall \bar{\alpha}. (a : \tau)
\]

In that particular case, constraint generation is in fact simpler:

\[
\llangle (a : \forall \bar{\alpha}. \tau) : \tau' \rrangle \equiv \forall \bar{\alpha}. \llangle a : \tau \rrangle \land (\forall \bar{\alpha}. \tau) \leq \tau'
\]

Exercise 39 Check this equivalence.
Examples  Consider the following two examples:

\[
\begin{align*}
& ((\exists \alpha. (\lambda x. x + 1 : \alpha \to \alpha)) : \text{int} \to \text{int}) & (\forall \alpha. (\lambda x. x + 1 : \alpha \to \alpha)) : \text{int} \to \text{int} \\
& \equiv \exists \alpha. ((\lambda x. x + 1 : \alpha \to \alpha) : \text{int} \to \text{int}) & \equiv \forall \alpha. ((\lambda x. x + 1 : \alpha \to \alpha) : \gamma)
\end{align*}
\]

\[
\begin{align*}
& \exists \alpha. (\alpha = \text{int}) & \equiv \forall \alpha. (\alpha = \text{int} \land \alpha \to \alpha = \gamma) \\
& \equiv \text{true} & \equiv \forall \alpha. \alpha = \text{int} \\
& \equiv \text{false}
\end{align*}
\]

The left-hand side example is well-typed: The system infers that \(\alpha\) must be \text{int}. Because \(\alpha\) is a local type variable, it does not appear in the final constraint. The right-hand side example is ill-typed: The system checks that \(\alpha\) is used in an abstract way, which is not the case here, since the code implicitly assumes that \(\alpha\) is \text{int}. By contrast, the following example is well-typed:

\[
\begin{align*}
& ((\forall \alpha. (\lambda x. x : \alpha \to \alpha)) : \text{int} \to \text{int}) \\
& = \forall \alpha. ((\lambda x. x : \alpha \to \alpha) : \gamma) \land \exists \alpha. ((\lambda x. x : \alpha \to \alpha) : \text{int} \to \text{int}) \\
& \equiv \forall \alpha. \exists \gamma. \alpha \to \alpha = \gamma \land \exists \alpha. \alpha = \text{int} \\
& \equiv \text{true}
\end{align*}
\]

The system checks that \(\alpha\) is used in an abstract way, which is indeed the case here. It also checks that, if \(\alpha\) is appropriately instantiated, the code admits the expected type \text{int} \to \text{int}.

The two next examples are similar and show the importance of the scope of existential variables. In the first one, the variable \(\alpha\) is bound outside the let construct:

\[
\begin{align*}
& ((\exists \alpha. (\text{let } f = (\lambda x. x : \alpha \to \alpha) \text{ in } (f 0, f \text{ true})) : \gamma) \\
& \equiv \exists \alpha. (\text{let } f : \alpha \to \alpha \text{ in } \exists \gamma_1 \gamma_2. (f \leq \text{int} \to \gamma_1 \land f \leq \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma)) \\
& \equiv \exists \alpha \gamma_1 \gamma_2. (\alpha \to \alpha = \text{int} \land \alpha \to \alpha = \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma) \\
& \equiv \exists \alpha. (\alpha = \text{int} \land \alpha = \text{bool}) \\
& \equiv \text{false}
\end{align*}
\]

Then \(f\) receives the monotype \(\alpha \to \alpha\) and the example is ill-typed. In the other example, \(\alpha\) is bound within the let construct:

\[
\begin{align*}
& ((\text{let } f = \exists \alpha. (\lambda x. x : \alpha \to \alpha) \text{ in } (f 0, f \text{ true}) : \gamma) \\
& \equiv \text{let } f : \forall \beta [\exists \alpha. (\alpha \to \alpha = \beta)]. \beta \text{ in } \exists \gamma_1 \gamma_2. (f \leq \text{int} \to \gamma_1 \land f \leq \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma) \\
& \equiv \text{let } f : \forall \alpha. \alpha \to \alpha \text{ in } \exists \gamma_1 \gamma_2. (\ldots) \\
& \equiv \exists \gamma_1 \gamma_2. (\text{int} = \gamma_1 \land \text{bool} = \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma) \\
& \equiv \text{int} \times \text{bool} = \gamma
\end{align*}
\]

Here, the term \(\exists \alpha. (\lambda x. x : \alpha \to \alpha)\) has the same principal type scheme as \(\lambda x. x\), namely \(\forall \alpha. \alpha \to \alpha\), which is the type scheme that \(f\) receives.

Type annotations in the real world  For historical reasons, type variables are not explicitly bound in OCaml. (Retrospectively, that’s bad!) They are implicitly existentially bound at the nearest enclosing toplevel let construct. In Standard ML, type variables are
implicitly *universally* bound at the nearest enclosing toplevel let construct. In Glasgow Haskell, type variables are implicitly existentially bound within patterns: ‘A pattern type signature brings into scope any type variables free in the signature that are not already in scope’ (Peyton Jones and Shields, 2004). Constraints help understand these varied design choices uniformly.

### 5.4.2 Polymorphic recursion

Recall below the typing rule \texttt{FixAbs} for recursive functions, which leads to the derived typing rule \texttt{LetRec} for recursive definitions:

\[
\begin{align*}
\text{FixAbs} & : \Gamma, f : \tau \vdash \lambda x. a : \tau \\
\text{LetRec} & : \Gamma, f : \bar{\alpha} \# \Gamma, \bar{a} \vdash a_2 : \tau_2 \\
& \quad \Gamma, f : \forall \bar{\alpha}. \tau_1 \vdash a_2 : \tau_2 \\
& \quad \Gamma \vdash \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau_2
\end{align*}
\]

These rules require occurrences of \( f \) to have *monomorphic type* within the recursive definition (that is, within \( \lambda x. a_1 \)). This is visible also in terms of type inference, as the two following constraints are equivalent:

\[
\langle \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau \rangle \equiv \text{let } f : \forall \alpha \beta [\langle \text{let } f : \alpha \to \beta; x : \alpha \text{ in } \langle a_1 : \beta \rangle \rangle]. \alpha \to \beta \text{ in } \langle a_2 : \tau \rangle
\]

On the right-hand side, all occurrences of \( f \) within \( a_1 \) have the same type \( \alpha \to \beta \). This is problematic in some situations, most particularly when defining functions over nested algebraic data types (Bird and Meertens, 1998; Okasaki, 1999).

This problem is solved by introducing *polymorphic recursion*, that is, by allowing \( \mu \)-bound variables to receive a polymorphic type scheme, using the following typing rules:

\[
\begin{align*}
\text{FixAbsPoly} & : \Gamma, f : \sigma \vdash \lambda x. a : \sigma \\
\text{LetRecPoly} & : \Gamma, f : \sigma \vdash \lambda x. a_1 : \sigma \quad \Gamma, f : \sigma \vdash a_2 : \tau \\
& \quad \Gamma \vdash \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau
\end{align*}
\]

This extension of ML is due to Mycroft (1984).

In System F, there is no problem to begin with; no extension is necessary. Polymorphic recursion alters, to some extent, Damas and Milner’s type system. Now, not only let-bound, but also \( \mu \)-bound variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed \( \lambda \)-calculus. This has two noticeable consequences: *monomorphization*, a technique employed in some ML compilers (Tolmach and Oliva, 1993; Cejin et al., 2007), is no longer possible; besides, *type inference* becomes problematic!

Type inference for ML with polymorphic recursion is undecidable (Henglein, 1993). It is equivalent to the undecidable problem of *semi-unification*. Yet, type inference in the presence of polymorphic recursion can be made simple by relying on a *mandatory type annotation*. 
The syntax and typing rules for recursive definitions become:

**FixAbsPoly**

\[ \Gamma, f : \sigma \vdash \lambda x. a : \sigma \]

\[ \Gamma \vdash \mu(f:\sigma).\lambda x. a : \sigma \]

**LetRecPoly**

\[ \Gamma, f : \sigma \vdash \lambda x. a_1 : \sigma \]

\[ \Gamma, f : \sigma \vdash a_2 : \tau \]

\[ \Gamma \vdash \text{let rec } (f:\sigma) = \lambda x. a_1 \text{ in } a_2 : \tau \]

The type scheme \(\sigma\) no longer has to be guessed. With this feature, contrary to what was said earlier (p. 109), type annotations are not just restrictive: they are sometimes required for type inference to succeed. The constraint generation rule becomes:

\[ \langle \text{let rec } (f:\sigma) = \lambda x. a_1 \text{ in } a_2 : \tau \rangle = \text{let } f : \sigma \text{ in } (\langle \lambda x. a_1 : \sigma \rangle \land \langle a_2 : \tau \rangle) \]

It is clear that \(f\) receives type scheme \(\sigma\) both inside and outside of the recursive definition.

### 5.4.3 Unification under a mixed prefix

*Unification under a mixed prefix* means unification in the presence of both existential and universal quantifiers. We extend the basic unification algorithm with support for universal quantification. The solved forms are unchanged: universal quantifiers are always eliminated.

In short, in order to reduce \(\forall \alpha. C\) to a solved form, where \(C\) is itself a solved form—see (Pottier and Rémy, 2003, p. 109) for details:

- If a rigid variable is equated with a constructed type, fail.
  For example, \(\forall \alpha. \exists \beta \gamma. (\alpha = \beta \rightarrow \gamma)\) is false.

- If two rigid variables are equated, fail.
  For example, \(\forall \alpha \beta. (\alpha = \beta)\) is false.

- If a free variable dominates a rigid variable, fail.
  For example, \(\forall \alpha. \exists \beta. (\gamma = \alpha \rightarrow \beta)\) is false.

- Otherwise, one can decompose \(C\) as \(\exists \beta. (C_1 \land C_2)\), where \(\alpha \beta \not\equiv C_1\) and \(\exists \beta. C_2 \equiv \text{true}\); in that case, \(\forall \alpha. C\) reduces to just \(C_1\).
  
  For example, \(\forall \alpha. \exists \beta \gamma_1 \gamma_2. (\beta = \alpha \rightarrow \gamma \land \gamma = \gamma_1 \rightarrow \gamma_2)\) reduces to just \(\exists \gamma_1 \gamma_2. (\gamma = \gamma_1 \rightarrow \gamma_2)\).
  
  The constraint \(\forall \alpha. \exists \beta. (\beta = \alpha \rightarrow \gamma)\) is equivalent to \text{true}.

OCaml implements a form of unification under a mixed prefix. This is illustrated by the following interactive OCaml session:

```
let module M : sig val id : 'a -> 'a end = struct let id x = x + 1 end in M.id

Values do not match: val id : int -> int is not included in val id : 'a -> 'a
```

This gives rise to a constraint of the form \(\forall \alpha. \alpha = \text{int}\), while the following example gives rise to a constraint of the form \(\exists \beta. \forall \alpha. \alpha = \beta\):
5.5 Equi- and iso-recursive types

Product and sum types alone do not allow describing data structures of unbounded size, such as lists and trees. Indeed, if the grammar of types is \( \tau ::= \text{unit} \mid \tau \times \tau \mid \tau + \tau \), then it is clear that every type describes a finite set of values. For every \( k \), the type of lists of length at most \( k \) is expressible using this grammar. However, the type of lists of unbounded length is not: “A list is either empty or a pair of an element and a list.” We need something like this:

\[
\text{list} \, \alpha \quad \triangleright \quad \text{unit} + \alpha \times \text{list} \, \alpha
\]

But what does \( \triangleright \) stand for? Is it equality, or some kind of isomorphism?

There are two standard approaches to recursive types, dubbed the equi-recursive and iso-recursive approaches. In the equi-recursive approach, a recursive type is equal to its unfolding. In the iso-recursive approach, a recursive type and its unfolding are related via explicit coercions.

5.5.1 Equi-recursive types

In the equi-recursive approach, the usual syntax of types:

\[
\tau ::= \alpha \mid F \bar{\tau}
\]

is no longer interpreted inductively. Instead, types are the regular trees built on top of this signature. If desired, it is possible to use finite syntax for recursive types:

\[
\tau ::= \alpha \mid \mu \alpha.(F \bar{\tau})
\]

We do not allow the seemingly more general \( \mu \alpha.\tau \), because \( \mu \alpha.\alpha \) is meaningless, and \( \mu \alpha.\beta \) or \( \mu \alpha.\mu \beta.\tau \) are useless. If we write \( \mu \alpha.\tau \), it should be understood that \( \tau \) is contractive, that is, \( \tau \) is a type constructor application. For instance, the type of lists of elements of type \( \alpha \) is:

\[
\mu \beta.(\text{unit} + \alpha \times \beta)
\]

Each type in this syntax denotes a unique regular tree, sometimes known as its infinite unfolding. Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to decide whether two types are equal, that is, have identical infinite unfoldings.

let r = ref (fun x -> x) in
let module M : sig val id : 'a -> 'a end = struct let id = !r end in M.id;;

Values do not match: val id : 'a -> 'a
is not included in val id : 'a -> 'a
This can be done efficiently, either via the algorithm for comparing two DFAs, or by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

One can also prove [Brandt and Henglein (1998)] that equality is the least congruence generated by the following two rules:

\[
\begin{align*}
\text{Fold/Unfold} & \quad \mu \alpha. \tau = [\alpha \mapsto \mu \alpha. \tau] \tau \\
\text{Uniqueness} & \quad \tau_1 = [\alpha \mapsto \tau_1] \tau \quad \tau_2 = [\alpha \mapsto \tau_2] \tau \\
& \quad \Rightarrow \tau_1 = \tau_2
\end{align*}
\]

In both rules, \( \tau \) must be contractive. This axiomatization does not directly lead to an efficient algorithm for deciding equality, though. In the presence of equi-recursive types, structural induction on types is no longer permitted—but we never used it anyway. It remains true that \( F \tilde{\tau}_1 = F \tilde{\tau}_2 \) implies \( \tilde{\tau}_1 = \tilde{\tau}_2 \)—this was used in our Subject Reduction proofs. It remains true that \( F_1 \tilde{\tau}_1 = F_2 \tilde{\tau}_2 \) implies \( F_1 = F_2 \)—this was used in our Progress proofs. So, the reasoning that leads to type soundness is unaffected.

**Exercise 40** Prove type soundness for the simply-typed \( \lambda \)-calculus in Coq. Then, change the syntax of types from Inductive to CoInductive.

How is type inference adapted for equi-recursive types? The syntax of constraints is unchanged: they remain systems of equations between finite first-order types, without \( \mu \)'s. Their interpretation changes: they are now interpreted in a universe of regular trees. As a result, constraint generation is unchanged; constraint solving is adapted by removing the occurs check.

**Exercise 41** Describe solved forms and show that every solved form is either false or satisfiable.

Here is a function that measures the length of a list:

\[
\mu(\text{length}). \lambda x. \text{case } x \text{ of } \lambda(). 0 \circ \lambda(y, z). 1 + \text{length } z
\]

Type inference gives rise to the cyclic equation \( \beta = \text{unit} + \alpha \times \beta \), where \text{length} has type \( \beta \rightarrow \text{int} \). That is, \text{length} has principal type scheme: \( \forall \alpha. (\mu \beta. \text{unit} + \alpha \times \beta) \rightarrow \text{int} \) or, equivalently, principal constrained type scheme: \( \forall \alpha[\beta = \text{unit} + \alpha \times \beta] \). \( \beta \rightarrow \text{int} \). The cyclic equation that characterizes lists was never provided by the programmer, but was inferred.

OCaml implements equi-recursive types upon explicit request, launching the interactive session with the command “ocaml -rectypes”:

\[
\begin{align*}
type ('a, 'b) sum = \text{Left of 'a | Right of 'b} \\
type ('a, 'b) sum = \text{Left of 'a | Right of 'b} \\
let rec length x = \text{function Left ()} \to 0 \mid \text{Right (y, z)} \to 1 + \text{length } z \\
val length : ((\text{unit, 'b } \times \text{ 'a}) \text{ sum as 'a}) \rightarrow \text{int} = (\text{fun})
\end{align*}
\]
Notice that `-rectypes` is only an option which is not on by default. Equi-recursive types are simple and powerful, but in practice, they are perhaps too expressive. Continuing with in the `-rectype` option:

```ocaml
let rec map f = function [] -> [] | y :: z -> map f y :: map f z

val map : 'a -> ('b list as 'b) -> ('c list as 'c) = ⟨fun⟩
```

This expression has type int but is used with type 'a list as 'a

```ocaml
map () ([]; [];[])
- : 'a list as 'a = ([]; [[]])
```

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error. Hence, by default, OCaml typechecker reject type cycles that do not involve an object type or a variant type. In a normal OCaml session (no `-rectypes`), the following is still accepted, though:

```ocaml
let f x = x#hello x;
val f : (<hello : 'a -> 'b; .. > as 'a) -> 'b = ⟨fun⟩
```

OCaml implements a partial occurs check that stops at object and variant types: equi-recursive types are allowed provided every infinite path crosses an object or a variant type.

### 5.5.2 Iso-recursive types

In the iso-recursive approach, the user is allowed to introduce new type constructors `D` via (possibly mutually recursive) declarations:

\[ D\tilde{\alpha} \approx \tau \]  
(where \( \text{ftv}(\tau) \subseteq \tilde{\alpha} \))

Each such declaration adds a unary constructor `fold_D` and a unary destructor `unfold_D` with the following types and the new reduction rule:

\[
fold_D : \forall \tilde{\alpha} . \tau \to D\tilde{\alpha} \quad \text{unfold}_D : \forall \tilde{\alpha} . D\tilde{\alpha} \to \tau \quad \text{unfold}_D (\text{fold}_D v) \to v
\]

Ideally, iso-recursive types should not have any runtime cost. One solution is to compile constructors and destructors away into a target language with equi-recursive types. Another solution is to see iso-recursive types as a restriction of equi-recursive types where the source language does not have equi-recursive types but instead two unary destructors `fold_D` and `unfold_D` with the semantics of the identity function. Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the **UNIQUENESS** typing rule.
For example, iso-recursive lists can be defined as follows. A parametrized, iso-recursive type of lists is:

$$\text{list } \alpha \approx \text{unit} + \alpha \times \text{list } \alpha.$$  

The empty list is:  

$$\text{fold}_{\text{list}} \left(\text{inj}_{1}()\right) : \forall \alpha. \text{list } \alpha.$$  

A function that measures the length of a list is:  

$$\mu(\text{length}) \cdot \lambda x. \text{case} \left(\text{unfold}_{\text{list}} x\right) \text{ of } \lambda(). 0 \circ \lambda(x, x). 1 + \text{length } x : \forall \alpha. \text{list } \alpha \rightarrow \text{int}$$

One folds upon construction and unfolds upon deconstruction.

In the iso-recursive approach, types remain finite. The type \text{list } \alpha is just an application of a type constructor to a type variable. As a result, type inference is unaffected. The occurs check remains.

### 5.5.3 Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums. This suppresses the verbosity of explicit folds and unfolds as well as the fragility and inconvenience of numeric indices—instead, named record fields and data constructors are used. For instance,  

$$\text{fold}_{\text{list}} \left(\text{inj}_{1}()\right) \text{ is replaced with } \text{Nil}()$$

An algebraic data type constructor \(D\) is introduced via a record type or variant type definition:

$$D \tilde{\alpha} \approx \prod_{\ell \in L} \ell : \tau_{\ell} \quad \text{or} \quad D \tilde{\alpha} \approx \sum_{\ell \in L} \ell : \tau_{\ell}$$

The set \(L\) denotes a finite set of record labels or data constructors \(\{\ell_{1} \ldots \ell_{n}\}\), which is fixed for a given definition. Algebraic data type definitions can be mutually recursive.

The record type definition \(D \tilde{\alpha} \approx \prod_{\ell \in L} \ell : \tau_{\ell}\) introduces a record \(n\)-ary constructor and \(n\) record unary destructors with the following types:

$$C := \ldots | (\ell_{1} = \cdot, \ldots \ell_{n} = \cdot) \quad d := \ldots | (\cdot.\ell_{1}) | \ldots (\cdot.\ell_{n})$$

$$\{\ell_{1} = \cdot, \ldots \ell_{n} = \cdot\} : \forall \tilde{\alpha}. \tau_{\ell_{1}} \rightarrow \ldots \tau_{\ell_{n}} \rightarrow D \tilde{\alpha} \quad \cdot.\ell : \forall \tilde{\alpha}. D \tilde{\alpha} \rightarrow \tau_{\ell}$$

The variant type definition \(D \tilde{\alpha} \approx \sum_{\ell \in L} \ell : \tau_{\ell}\) introduces unary variant constructors and variant destructor of arity \(n + 1\) with the following types:

$$C := \ldots | (\ell) \quad d := \ldots | \text{case} \cdot \text{ of } [\ell_{1} : \cdot \ldots \ell_{n} : \cdot] \quad \ell : \forall \tilde{\alpha}. \tau_{\ell} \rightarrow D \tilde{\alpha}$$

$$\text{case} \cdot \text{ of } [\ell_{1} : \cdot \ldots \ell_{n} : \cdot] : \forall \tilde{\alpha}. \beta. D \tilde{\alpha} \rightarrow (\tau_{\ell_{1}} \rightarrow \beta) \rightarrow \ldots (\tau_{\ell_{n}} \rightarrow \beta) \rightarrow \beta$$

For example, an algebraic data type of lists is \(\text{list } \alpha \approx \text{Nil} : \text{unit} + \text{Cons} : \alpha \times \text{list } \alpha\) gives rise to:

$$\text{case} \cdot \text{ of } [\text{Nil} : \cdot \ldots \text{Cons} : \cdot] : \forall \alpha. \beta. \text{list } \alpha \rightarrow (\text{unit} \rightarrow \beta) \rightarrow ((\alpha \times \text{list } \alpha) \rightarrow \beta) \rightarrow \beta$$

$$\text{Nil} : \forall \alpha. \text{unit} \rightarrow \text{list } \alpha$$

$$\text{Cons} : \forall \alpha. (\alpha \times \text{list } \alpha) \rightarrow \text{list } \alpha$$
A function that measures the length of a list is:
\[
\mu(\text{length}) \cdot \lambda x. \text{case } x \text{ of } \text{Nil} : \lambda() \circ \text{Cons} : \lambda(y, z). 1 + \text{length } z : \forall \alpha. \text{list } \alpha \rightarrow \text{int}
\]

**Mutable record fields** In OCaml, a record field can be marked *mutable*. This introduces an extra binary destructor for writing this field: \((\cdot \ell \leftarrow \cdot)\) of type \(\forall \alpha. D \rightarrow \tau \rightarrow \text{unit}\). However, this also makes record construction a destructor since, when fully applied it is *not a value* but it allocates a piece of store and returns its location. Thus, due to the value restriction, the type of such expressions cannot be generalized.

## 5.6  **HM(X)**

Soundness and completeness of type inference are in fact easier to prove if one adopts a *constraint-based specification* of the type system, as in the language HM(X) introduced by Odersky et al. (1999).

In HM(X), judgments take the form \(C, \Gamma \vdash a : \tau\), called a constrained typing judgments. Read under the assumption \(C\) and typing environment \(\Gamma\), the program \(a\) has type \(\tau\). Here \(C\) constrains free type variables of the judgment while \(\Gamma\) provides the type of free program variables of \(a\). The constraint \(C\) ranges over first-order typing constraints—except that we require type constraints to have no free program variables. In a constrained typing judgment \(C, \Gamma \vdash a : \tau\),

The parameter \(X\) in HM(X) stands for the logic of the constraint language. We have so far only consider constraints with an equality predicate. However, the equality replaced may be by an asymmetric subtyping predicate \(\leq\), which makes the language of constraints richer.

The typing rules also use an entailment predicate \(C \vdash C'\) between constraints that is more general than constraint equivalence. Entailment is defined as expected: \(C \vdash C'\) if and only if any ground assignment that satisfies \(C\) also satisfies \(C'\).
Typing rules for \( \text{HM}(X) \) are presented in Figure 5.7. Moreover, judgment are taken up to constraint equivalence. The constraint \( \exists \sigma \) in the premise of Rule \( \text{HM-VAR} \) is an abbreviation for \( \exists \alpha. C_0 \) where \( \sigma \) is \( \forall \alpha[C_0]\). \( \tau \). A valid judgment is one that has a derivation with those typing rules. In a valid judgment, \( C \) may not be satisfiable. A program is well-typed in environment \( \Gamma \) if it has a valid judgment \( C, \Gamma \vdash a : \tau \) for some \( \tau \) and satisfiable constraint \( C \).

When considering equality only constraints, \( \text{HM}(=) \) is in fact equivalent to \( \text{ML} \): if \( \Gamma \) and \( \tau \) contain only Damas-Milner’s type schemes, then \( \Gamma \vdash a : \tau \) in \( \text{ML} \) if and only if \( \text{true, } \Gamma \vdash a : \tau \) in \( \text{HM}(X) \). Moreover, if \( C, \Gamma \vdash a : \tau \) in \( \text{HM}(X) \) and \( \varphi \) is an idempotent solution of \( C \), we have \( \text{true, } \Gamma \varphi \vdash a : \tau \varphi \) in \( \text{HM}(X) \) where \( (\cdot)\varphi \) translates \( \text{HM}(X) \) type schemes into \( \text{ML} \) type schemes—applying the substitution \( \varphi \) on the fly.

As for \( \text{ML} \), there is an equivalent syntax-directed presentation of the typing rules. However, we may take advantage of program variables in constraints to go one step further and mix the constraint \( C \) (without free program variables) and the typing environment \( \Gamma \) into a single constraint \( C \) now with possibly free program variables. Judgments take the form \( C \vdash a : \tau \) where \( C \) constrains type variables and assign constrained type schemes to program variables. The type system, called \( \text{PCB}(X) \), is described on Figure 5.8. It is equivalent to \( \text{HM}(X) \) (see Pottier and Rémy, 2005) for the precise comparison.

For example of a derivation in \( \text{PCB}(X) \), let \( a \) be \( \text{let } y = \lambda x. x \text{ in } y y \):

\[
\begin{align*}
&\text{VAR} \quad \text{VAR} \quad \text{VAR} \\
&\text{let } x : \alpha_0 \text{ in } x \leq \alpha \vdash x : \alpha & y \leq \beta_2 \to \beta_1 \vdash y : \beta_2 & y \leq \beta_2 \vdash y : \beta_1 \\
&\text{fun} \quad \text{app} \quad \text{fun} \\
&\quad \text{let } x : \alpha_0 \text{ in } x \leq \alpha \vdash \lambda x. x : \alpha_0 \to \alpha & y \leq \beta_2 \to \beta_1 \vdash y : \beta_2 & y \leq \beta_2 \vdash y : \beta_1 \\
&\text{let} \quad \text{app} \quad \text{let} \\
&\quad \text{exists} \quad \exists \beta_2. C \vdash a : \beta_1
\end{align*}
\]

where \( C \) is

\[\text{let } y : \forall \alpha \alpha_0 [\text{let } x : \alpha_0 \text{ in } x \leq \alpha], \alpha_0 \to \alpha \text{ in } y \leq \beta_2 \to \beta_1 \land y \leq \beta_2\]

The constraint \( C \) can be simplified as follows:

\[
\exists \beta_2, C = \exists \beta_2, \text{let } y : \forall \alpha. \alpha_0 = \alpha, \alpha_0 \to \alpha \text{ in } y \leq \beta_2 \to \beta_1 \land y \leq \beta_2
\]

\[\equiv \exists \beta_2, \text{let } y : \forall \alpha. \alpha \to \alpha \text{ in } y \leq \beta_2 \to \beta_1 \land y \leq \beta_2
\]

\[\equiv \exists \beta_2 \alpha_1 \alpha_2, \alpha_1 \to \alpha_1 = \beta_2 \to \beta_1 \land \alpha_2 \to \alpha_2 = \beta_2
\]

\[\equiv \exists \alpha. \beta_1 = \alpha \to \alpha
\]

Hence, we also have \( \exists \alpha. \beta_1 = \alpha \to a : \beta_1 \). This is a valid judgment, but not a satisfiable one. However, by rule [\text{PCB-SUB}] and [\text{PCB-EXISTS}] we have \( \exists \beta_1, (\exists \alpha. \beta_1 \alpha \to \alpha) \land \beta_1 = \beta \to \beta \vdash a : \beta \to \beta \), which is equivalent to \( \text{true, } \Gamma \vdash a : \tau \) and is both valid and satisfiable.

The type inference algorithm for \( \text{ML} \) is sound and complete for \( \text{PCB}(X) \):

- **Soundness**: \( \langle a : \tau \rangle \vdash a : \tau \). The constraint inferred for a typing validates the typing.

- **Completeness**: If \( C \vdash a : \tau \) then \( \Gamma \vdash \langle a : \tau \rangle \). The constraint inferred for a typing is more general than any constraint that validates the typing.
5.7. TYPE RECONSTRUCTION IN SYSTEM F

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCB- VAR</td>
<td>( C \vdash x \leq \tau )</td>
<td>( C \vdash x : \tau )</td>
</tr>
<tr>
<td>PCB- ABS</td>
<td>( C \vdash a : \tau )</td>
<td>let ( x : \tau_0 ) in ( C \vdash a : \tau_0 \rightarrow \tau )</td>
</tr>
<tr>
<td>PCB- APP</td>
<td>( C_1 \vdash a_1 : \tau_2 \rightarrow \tau_1 )  ( C_2 \vdash a_2 : \tau_2 )  ( C_1 \land C_2 \vdash a_1 \ a_2 : \tau_1 )</td>
<td>( C_1 \vdash a_1 : \tau_2 \rightarrow \tau_1 )  ( C_2 \vdash a_2 : \tau_2 )  ( C_1 \land C_2 \vdash a_1 \ a_2 : \tau_1 )</td>
</tr>
<tr>
<td>PCB- LET</td>
<td>( C_1 \vdash a_1 : \tau_1 )  ( C_2 \vdash a_2 : \tau_2 )  ( \text{let } x : \forall \nu[C_1], \tau_1 \text{ in } C_2 \vdash x = a_1 \text{ in } a_2 : \tau_2 )</td>
<td>( C_1 \vdash a_1 : \tau_1 )  ( C_2 \vdash a_2 : \tau_2 )  ( \text{let } x = a_1 \text{ in } a_2 : \tau_2 )</td>
</tr>
<tr>
<td>PCB- SUB</td>
<td>( C \vdash a : \tau_1 )  ( C \land \tau_1 \leq \tau_2 \vdash a : \tau_2 )  ( \exists \alpha. C \vdash a : \tau )</td>
<td>( C \vdash a : \tau_1 )  ( C \land \tau_1 \leq \tau_2 \vdash a : \tau_2 )  ( \exists \alpha. C \vdash a : \tau )</td>
</tr>
</tbody>
</table>

Figure 5.8: Typing rules for PCB(X)

Note Our presentation of \( \text{HM}(X) \) is incomplete. See also Skalka and Pottier (2002) for a more recent presentation of \( \text{HM}(X) \) and Pottier and Rémy (2005) for a detailed presentation of several variants of \( \text{HM}(X) \).

Our proof of type soundness for ML only applies for \( \text{HM} (=) \). One may prove type soundness for \( \text{HM}(X) \) in the general case for some logic \( X \), under the axiom that the arrow type constructor is contra-variant for subtyping. See Pottier and Rémy (2005).

5.7 Type reconstruction in System F

Type checking in explicitly-typed System F is easy. Still, an implementation must carefully deal with variable bindings and renaming when applying type substitutions. However, as we have seen, programming with fully-explicit types is unpractical.

Full type inference in System F has long been an open problem, until Wells (1999) proved it undecidable by showing that it is equivalent to the semi-unification problem which was earlier proved undecidable. (Notice that the full type-inference problem is not directly related to second-order unification but rather to semi-unification.)

Hence, we must perform partial type inference in System F. Either type inference is incomplete, or some amount of type annotations must be provided. Several solutions are used in practice. They alleviate the need for a lot of redundant type annotations.

5.7.1 Type inference based on Second-order unification

Full type inference is equivalent to semi-unification. However, type inference becomes equivalent to second-order unification if all the positions of type abstractions and type applications are explicit, while types are themselves left implicit. That is, if terms are

\[ M ::= \text{x} \mid \lambda x:\alpha. M \mid M \cdot M \mid \Lambda. M \mid M ? \]

where the question marks stand for type variables and types to be inferred. Although, the problem of second-order unification is undecidable, there are semi-algorithms that often work well in common cases. This method was proposed by Pfenning (1988).
In fact, partial type inference based on second-order unification can be mixed with type checking. Explicit polymorphism may be reintroduced as in explicitly-typed System F while explicitly-controlled implicit instantiation can be performed as above by second-order unification. The source language is:

\[ M := x | \lambda x : \tau. M | M M | \Lambda \alpha . M | M \tau | \lambda x : \? . M | M \? | \text{let } f = \Lambda \alpha_1 \ldots \Lambda \alpha_n . M \text{ in } M \]

The new let-binding form is used to declare type arguments that will be made implicit. Then, every occurrence of such a variable automatically adds type-application holes at the corresponding positions and type parameters will be inferred using second-order unification. This amounts to understanding the new let-binding form as follows:

\[ \text{let } f = \Lambda \alpha_1 \ldots \Lambda \alpha_n . M_1 \text{ in } M_2 \triangleq \text{let } f = \Lambda \alpha_1 \ldots \Lambda \alpha_n . M_1 \text{ in } [f \mapsto f \? \ldots ? ] M_2 \]

Type inference in this language still reduces to second-order unification.

### 5.7.2 Bidirectional type inference

Type-checking in explicit simply-typed \( \lambda \)-calculus is easy because typing rules have an algorithmic reading. This implies that they are syntax directed, but also that judgments can be read as functions where some arguments are inputs and others are output. In the implicit calculus, the rules are still syntax-directed, but some of them do not have an obvious algorithmic reading. Typically, \( \Gamma \) and \( a \) would be inputs and \( \tau \) is an output in the judgment \( \Gamma \vdash a : \tau \), which we may represent as \( \Gamma^\dagger \vdash a^\dagger : \tau^\dagger \). However, in the rule for abstraction:

\[
\text{Abs} \quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x . a : \tau_0 \rightarrow \tau}
\]

the type \( \tau_0 \) is used both as input (in the premise) and as an output in the conclusion. Hence, type-checking the implicit simply-typed \( \lambda \)-calculus is not straightforward. In some cases, the type of the function may be known, e.g. when the function is an argument to an expression of a known type. Then, it suffices to check the proposed type is indeed correct.

Formally, we need algorithmic reading of the typing judgment, depending on whether the return type is known or unknown. We may split the typing judgment \( \Gamma \vdash a : \tau \) into two
judgments $\Gamma \vdash a \downarrow \tau$ to check that $a$ may be assigned the type $\tau$ and $\Gamma \vdash a \uparrow \tau$ to infer the type $\tau$ of $a$ (or with information flows $\Gamma^\uparrow \vdash a^\downarrow \tau^\downarrow$ and $\Gamma^\uparrow \vdash a^\uparrow \tau^\uparrow$. Both judgments are recursively defined by the rules of Figure ??: the checking mode can call the inference mode when needed; conversely, annotations may be used to turn inference mode into checking mode. (As a particular case, annotations on type abstractions enable the inference mode.)

An example of bidirectional derivation is given on Figure 5.10. The type $\tau$ stands for $(\tau_1 \rightarrow \tau_1) \rightarrow \tau_2$ and the environment $\Gamma$ is $f : \tau$.

The bidirectional method can be extended to deal with polymorphic types, but it is more complicated. The idea, due to Cardelli (1993), was popularized by Pierce and Turner (2000), and Odersky et al. (2001) and is still being improved Dunfield (2009).

**Predicative polymorphism**  
Predictive polymorphism is an interesting subcase of bidirectional type inference in the presence of predicative polymorphism. Predicative polymorphism is a restriction of impredicative polymorphism as can be found in System F. With predicative polymorphism, types are stratified so that polymorphic types can only be instantiated with simple types.

Interestingly, partial type inference can then still reduced to typing constraints under a mixed prefix (Rényi, 2005; Jones et al., 2006). Unfortunately, predicative polymorphism is too restrictive for use in programming languages: as polymorphic values often need to be put in data-structures whose constructors are polymorphic but impredicative polymorphism does not allow implicit instantiation of polymorphic constructors by polymorphic types.

One may also use a hierarchy of types where polymorphic types of rank $n$ can be instantiated with polymorphic types of a strictly lower rank. This increases expressiveness but $F$ is still more expressive than the union of all $F^n$.

Type inference with first-order constraints does not work for higher ranks.

**Local type inference**  
A simpler approach than *global* bidirectional type inference proposed by Pierce and Turner and improved by Odersky et al. is to perform bidirectional type inference *locally*, i.e. by considering for each node only a small context surrounding it.
Subtyping  Interestingly, bidirectional type inference can easily be extended to work in the presence of subtyping, which is not the case for methods based on second order unification.

5.7.3 Partial type inference in MLF

The language MLF [Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008] is an extension of System F especially designed for partial type inference—in fact for type inference a la ML within System F. That is, the inference algorithm performs first-order unification and aggressive ML-style let-generalization, but in the presence of second-order types. Interestingly, only parameters of functions that are used polymorphically need to be annotated in MLF; type abstractions and type annotation are always left implicit. However, for the purpose of type inference, MLF introduces richer types that enable to write “more principal types”, but that are also harder to read. The type inference method for MLF can be seen as a generalization of the constraint-based type inference for ML that handles polymorphic types.

5.8 Proofs and Solution to Exercises

Proof of Theorem 15

We proof \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \) if and only if \( \phi \vdash a : \phi \tau \) by induction on \( a \). We prove both implications independently because reasoning with equivalence is error-prone, since the arguments are similar but often not quite the same in both directions. The proof is thus a bit lengthy, but all cases are easy.

Case \( a \) is \( x \): Assume \( \phi \vdash a : \phi \tau \). By inversion of typing, this judgment must be derived by rule \[ \text{VAR} \]. Hence, \( \phi \tau = \phi \Gamma(x) \). By definition of satisfiability this implies \( \phi \vdash \tau = \Gamma(x) \). By definition of typing constraint, this is \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \).

Conversely, assume \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \). By definition of typing constraint, this is \( \phi \vdash \tau = \Gamma(x) \). By inversion of satisfiability we must have \( \phi \tau = \phi \Gamma(x) \). Hence, by rule \[ \text{VAR} \] we have \( \phi \Gamma \vdash a : \phi \tau \).

Case \( a \) is \( a_1 a_2 \): Assume \( \phi \Gamma \vdash a : \phi \tau \). By rule \[ \text{APP} \] there exists \( \tau_2 \) such that \( \phi \Gamma \vdash a_1 : \tau_2 \to \phi \tau \) and \( \phi \Gamma \vdash a_2 : \tau_2 \). Let \( \beta \not\equiv \Gamma \) and \( \phi' \) be \( \phi, \beta \mapsto \tau_2 \). We have \( \phi' \Gamma \vdash a_1 : \phi' \beta \to \tau \) and \( \phi' \Gamma \vdash a_2 : \beta \). Hence, by induction hypothesis \( \phi' \vdash \langle \Gamma \vdash a_1 : \beta \to \tau \rangle \) and \( \phi' \vdash \langle \Gamma \vdash a_2 : \beta \rangle \). Thus, \( \phi \vdash \exists \beta. \langle \Gamma \vdash a_1 : \beta \to \tau \rangle \land \langle \Gamma \vdash a_2 : \beta \rangle \). i.e. \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \).

Conversely, assume \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \). We have \( \phi \vdash \exists \beta. \langle \Gamma \vdash a_2 : \beta \rangle \land \langle \Gamma \vdash a_1 : \beta \to \tau \rangle \). We may assume \( w.l.o.g. \) that \( \beta \not\equiv \phi \). There must exist \( \phi' \) of the form \( \phi, \beta \mapsto \tau_2 \) such that \( \phi' \vdash \langle \Gamma \vdash a_2 : \beta \rangle \land \langle \Gamma \vdash a_1 : \beta \to \tau \rangle \). By induction hypothesis, this implies \( \phi \Gamma \vdash a_2 : \phi' \beta \) and \( \phi \Gamma \vdash a_1 : \phi' \beta \to \tau \), i.e. \( \phi \Gamma \vdash a_2 : \tau_2 \) and \( \phi \Gamma \vdash a_1 : \phi \tau_2 \to \tau \). By rule \[ \text{APP} \] we have \( \phi \Gamma \vdash a_1 a_2 : \phi \tau \).
5.8. PROOFS AND SOLUTION TO EXERCISES

Case a is $\lambda x.a_1$: Assume $\phi \Gamma \vdash a : \phi \tau$. We may assume w.l.o.g. that $x \not\in \Gamma$. By rule FUN there must exist $\tau_1$ and $\tau_2$ such that $\phi \Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi \tau = \tau_2 \rightarrow \tau_1$. Let $\beta_1$ and $\beta_2$ be disjoint from $\Gamma$ and $\phi'$ be $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$. Then, both $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi' \beta_1$ and $\phi' \tau = \phi'(\beta_2 \rightarrow \beta_1)$ hold. By induction hypothesis, $\phi' \vdash \langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle$ and $\phi' \vdash \tau = \beta_2 \rightarrow \beta_1$. Therefore, $\phi \vdash \exists \beta_1 \beta_2. \langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle \land \tau = \beta_2 \rightarrow \beta_1$. That is, $\phi \vdash \langle \Gamma \vdash a : \tau \rangle$.

Conversely, assume $\phi \vdash \langle \Gamma \vdash a : \tau \rangle$. By definition of constraints, we have $\phi \vdash \exists \beta_1 \beta_2. \langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle \land \tau = \beta_2 \rightarrow \beta_1$ for some $x$ disjoint from $\Gamma$. We may assume w.l.o.g. that $\beta_1, \beta_2 \not\in \phi$. There must exist $\phi'$ of the form $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$ such that $\phi' \vdash \langle \Gamma, x : \beta_2 \vdash a_1 : \tau_1 \rangle$ and $\phi' \vdash \tau = \beta_2 \rightarrow \beta_1$. By induction hypothesis, $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi' \beta_1$ and $\phi' \tau = \phi'(\beta_2 \rightarrow \beta_1)$. That is, $\phi \Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi \tau = \tau_2 \rightarrow \tau_1$. Hence, by rule FUN we have $\phi \Gamma \vdash a : \phi \tau$.

Solution of Exercise 36

See Bjørner (1994).

Solution of Exercise 37

Consider the module \textbf{struct f = let f = $\lambda x.x$ in f f end}. In core ML, the expression has principal type $\alpha \rightarrow \alpha$—but $\alpha$ cannot be generalized. Hence, $\textbf{sig f : } \forall \alpha. \alpha \rightarrow \alpha \textbf{ end}$ is not a signature for this module; nor is $\textbf{sig f : } \alpha \rightarrow \alpha \textbf{ end}$ since it is not a well-formed one. Correct signatures are $\textbf{sig f : } \tau \rightarrow \tau \textbf{ end}$ for any $\tau$, but they do not have a best element.
Chapter 6

Existential types

Compilation is type-preserving when each intermediate language is *explicitly typed*, and each compilation phase transforms a typed program into a typed program in the next intermediate language.

Type preserving compilation is interesting for several reasons: it can help debug the compiler; types can be used to drive optimizations; types can also be used to produce *proof-carrying code*; proving that types are preserved during compilation can be the first step towards proving that the *semantics* is preserved [Chlipala (2007)].

Besides, type-preserving compilation is quite challenging as it exhibits an encoding of programming constructs into programming language that usually requires richer type systems. Sometimes, an encoding later becomes a programming idiom that is used directly in the source language. There are several examples: closure conversion requires an extension of the language with existential types, which happens to very useful on their own. Closures are themselves a simple form of objects. Defunctionalization may be done manually on some particular programs, *e.g.* in web applications to monitor the computation.

A classic paper by Morrisett *et al.* [1999] shows how to go from System F to “Typed Assembly Language”, while preserving types along the way. Its main passes are:

1. *CPS conversion* fixes the order of evaluation, names intermediate computations, and makes all function calls tail calls;

2. *closure conversion* makes environments and closures explicit, and produces a program where all functions are closed;

3. allocation and initialization of tuples is made explicit;

4. the calling convention is made explicit, and variables are replaced with (an unbounded number of) machine registers.

127
In general, a type-preserving compilation phase involves not only a translation of terms, mapping $M$ to $\llbracket M \rrbracket$, but also a translation of types, mapping $\tau$ to $\llbracket \tau \rrbracket$, with the property:

$$\Gamma \vdash M : \tau \implies [\Gamma] \vdash [M] : [\tau]$$

The translation of types carries a lot of information: examining it is often enough to guess what the translation of terms will be.

### 6.1 Towards typed closure conversion

First-class functions may appear in the body of other functions. Hence, their own body may contain free variables that will be bound to values during the evaluation in the execution environment. Because they can be returned as values, and thus used outside of their definition environment, they must store their execution environment in their value. A closure is the packaging of the code of a first-class function with its runtime environment, so that it becomes closed, i.e. independent of the runtime environment and can be passed to another function and applied in another runtime environment. Closures can also be used to represent recursive functions and objects in the object-as-record-of-methods paradigm.

In the following, the source calculus has unary $\lambda$-abstractions, which can have free variables, while the target calculus has binary $\lambda$-abstractions, which must be closed. In the target language, we also use pattern matching over tuples. The translation will be naive, insofar as it will not handle functions of multiple arguments in a special way. One could argue that this is a feature, not a limitation, and that “uncurrying” (if desired) should be a separate type-preserving pass anyway. But closure conversion can also be easily extended to n-ary functions.

There are at least two variants of closure conversion: In the closure-passing variant, the closure and the environment are a single memory block; In the environment-passing variant, the environment is a separate block, to which the closure points. The impact of this choice on the term translations is minor. Closure-passing better supports simple recursive functions; but this is less obvious with mutually recursive ones. Closure-passing optimizes the case of closed functions: they do not need to create a closure—the code pointer can be passed directly. However, its impact on the type translations is more important: the closure-passing variant requires more type-theoretic machinery (recursive types and rows).

The closure-passing variant is as follows:

$$\llbracket \lambda x. M \rrbracket = \begin{array}{l}
\text{let } code = \lambda (clo, x).\text{let } (\_, x_1, \ldots, x_n) = clo \text{ in } [M] \text{ in } \\
(code, x_1, \ldots, x_n) \end{array}$$

$$\llbracket M_1 M_2 \rrbracket = \begin{array}{l}
\text{let } clo = [M_1] \text{ in } \\
\text{let } code = proj_0 clo \text{ in } \\
code (clo, [M_2]) \end{array}$$
6.1. TOWARDS TYPED CLOSURE CONVERSION

where \( \{ x_1, \ldots, x_n \} \) is \( \text{fv}(\lambda x. M) \) (the variables \textit{code} and \textit{clo} must be suitably fresh). Note that the layout of the environment must be known only at the closure allocation site, not at the call site. In particular, \( \text{proj}_0 \ \text{clo} \) need not know the size of \textit{clo}.

The environment-passing variant is as follows:

\[
\llbracket \lambda x. M \rrbracket = \ \text{let code} = \lambda(\textit{env}, x). \text{let } (x_1, \ldots, x_n) = \textit{env} in \llbracket M \rrbracket \text{ in }
\text{(code, } (x_1, \ldots, x_n) \text{)}
\]

\[
\llbracket M_1 \ \text{M}_2 \rrbracket = \ \text{let } (\text{code, env}) = \llbracket M_1 \rrbracket \text{ in }
\text{code (env, } \llbracket M_2 \rrbracket \text{)}
\]

where \( \{ x_1, \ldots, x_n \} = \text{fv}(\lambda x. M) \).

To understand type-preserving closure conversion, let us first focus on the environment-passing variant. How can closure conversion be made \textit{type-preserving}? The key issue is to find a sensible definition of the type translation. In particular, what is the translation of a function type, \( \llbracket \tau_1 \rightarrow \tau_2 \rrbracket \)? Let us examine the closure allocation code again. Suppose \( \Gamma \vdash \lambda x. M : \tau_1 \rightarrow \tau_2 \). Suppose, without loss of generality (see Remark 5), that \( \text{dom}(\Gamma) \) is exactly \( \text{fv}(\lambda x. M) \), \textit{i.e.} \( \{ x_1, \ldots, x_n \} \). Overloading the notation, if \( \Gamma = x_1 : \tau_1; \ldots ; x_n : \tau_n \), we also write \( \llbracket \Gamma \rrbracket \) for the tuple type \( \llbracket \tau_1 \rrbracket \times \ldots \times \llbracket \tau_n \rrbracket \). By hypothesis, we have \( \llbracket \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket \vdash \llbracket M \rrbracket : \llbracket \tau_2 \rrbracket \), so \textit{env} has type \( \llbracket \Gamma \rrbracket \), \textit{code} has type \( (((\llbracket \Gamma \rrbracket \times \llbracket \tau_1 \rrbracket ) \rightarrow \llbracket \tau_2 \rrbracket ) \times \llbracket \Gamma \rrbracket ) \). An existential quantification over the type of the environment abstracts away the differences in size and layout. Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable \( \alpha \) occur twice on the right-hand side.

The existential quantification also provides a form of \textit{security}. The caller cannot do anything with the environment except pass it as an argument to the code. In particular, it cannot inspect or modify the environment. For instance, in the source language, the following coding style guarantees that \( x \) remains even, no matter how \textit{f} is used:

\[
\text{let } f = \text{let } x = \text{ref } 0 \text{ in } \lambda() . x := (\text{!} x + 2); ! x
\]

After closure conversion, the reference \( x \) is reachable via the closure of \( f \). A malicious, untyped client could write an odd value to \( x \). However, a \textit{well-typed} client is unable to do so. This encoding is not just type-preserving, but also \textit{fully abstract}: it preserves (a typed
version of) observational equivalence (Ahmed and Blume, 2008).

**Remark 5** In order to support the hypothesis \( \text{dom}(\Gamma) = \text{fv}(\lambda x. M) \) at every \( \lambda \)-abstraction, it is possible to introduce an (admissible) weakening rule:

\[
\begin{array}{c}
\text{WEAKENING} \\
\Gamma_1; \Gamma_2 \vdash M : \tau \\
\hline
\Gamma_1; x : \tau'; \Gamma_2 \vdash M : \tau \\
\end{array}
\]

If the weakening rule is applied eagerly at every \( \lambda \)-abstraction, then the hypothesis is met, and closures have *minimal environments*. (In some cases, one may not use minimal environments, e.g. to allow sharing of environments between several closures.)

### 6.2 Existential types

One can extend System \( \mathsf{F} \) with *existential types*, in addition to universals:

\[
\tau ::= \ldots | \exists \alpha. \tau
\]

As in the case of universals, there are *type-passing* and *type-erasing* interpretations of the terms and typing rules and, in the latter interpretation, there are *explicit* and *implicit* versions. Let us first look at the type-erasing interpretation with an explicit notation for introducing and eliminating existential types.

#### 6.2.1 Existential types in Church style (explicitly typed)

The existential quantifier are introduced and eliminated as follows:

\[
\begin{array}{c}
\text{PACK} \quad \Gamma \vdash M : [\alpha \mapsto \tau']\tau \\
\hline
\Gamma \vdash \text{pack } \tau', M \text{ as } \exists \alpha. \tau : \exists \alpha. \tau
\end{array}
\]

\[
\begin{array}{c}
\text{UNPACK} \quad \Gamma \vdash M_1 : \exists \alpha. \tau_1 \\
\hline
\Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \\
\Gamma \vdash \text{let } \alpha, x = \text{unpack } M_1 \text{ in } M_2 : \tau_2
\end{array}
\]

The side condition \( \alpha \neq \tau_2 \) is *mandatory* here to ensure well-formedness of the conclusion. If well-formedness conditions were explicit in judgments, this could be equivalently defined as \( \Gamma \vdash \tau_2 \), as it would imply \( \alpha \neq \tau_2 \) since the last premise implies \( \alpha \neq \Gamma \).

Notice the *imperfect* duality between existential and universals, reminded below:

\[
\begin{array}{c}
\text{TAbs} \quad \Gamma, \alpha \vdash M : \tau \\
\hline
\Gamma \vdash \Lambda \alpha. M : \forall \alpha. \tau
\end{array}
\]

\[
\begin{array}{c}
\text{TApp} \quad \Gamma \vdash M : \forall \alpha. \tau \\
\hline
\Gamma \vdash M \tau' : [\alpha \mapsto \tau']\tau
\end{array}
\]

This suggests a simpler elimination form, perhaps like this:

\[
\begin{array}{c}
\Gamma \vdash M : \exists \alpha. \tau \\
\hline
\Gamma, \alpha \vdash \text{unpack } M : \tau
\end{array}
\]

*Broken!*
Informally, this could mean that, if $M$ has type $\tau$ for some unknown $\alpha$, then it has type $\tau$, where $\alpha$ is “fresh”. Unfortunately, this is a broken rule, as we could immediately universally quantify over $\alpha$ and conclude that $\Gamma \vdash M : \forall \alpha. \tau$. This is nonsense! Replacing the premise $\Gamma, \alpha \vdash M : \exists \alpha. \tau$ by the conjunction $\Gamma \vdash M : \exists \alpha. \tau$ and $\alpha \in \text{dom}(\Gamma)$ would make the rule even more permissive, so it wouldn’t help.

A correct elimination rule must force the existential package to be used in a way that does not rely on the value of $\alpha$. Hence, the elimination rule must have control over the user or continuation of the package—that is, over the term $M_2$. The restriction $\alpha \not\equiv \tau_2$ prevents writing “let $\alpha, x = \text{unpack} M_1$ in $x$”, which would be equivalent to the unsound “unpack $M$” discussed above. The fact that $\alpha$ is bound within $M_2$ forces it to be treated abstractly. In fact, $M_2$ must be polymorphic in $\alpha$. The rule could be written:

$$
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash \Lambda \alpha. \lambda x. M_2 : \forall \alpha. \tau_1 \to \tau_2 \quad \alpha \not\equiv \tau_2
$$

Or, more economically:

$$
\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma \vdash M_0 : \forall \alpha. \tau_1 \to \tau_2 \quad \alpha \not\equiv \tau_2
$$

$$
\Gamma \vdash \text{unpack} M_1 M_0 : \tau_2
$$

where $M_0$ would evaluate to a value of the form $\Lambda \alpha. \lambda x. M_2$.

One could even view “$\text{unpack}_{\exists \alpha. \tau}$” as a constant, of type $\exists \alpha. \tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta)$. The variable $\beta$, which stands for $\tau_2$, is bound prior to $\alpha$, so it naturally cannot be instantiated to a type that refers to $\alpha$. This reflects the side condition $\alpha \not\equiv \tau_2$. If desired, “$\text{pack}_{\exists \alpha. \tau}$” could also be viewed as a constant of type $\forall \alpha. (\tau \to \exists \alpha. \tau)$.

In summary, System F with existential types can also be presented as follows:

$$
\text{pack}_{\exists \alpha. \tau} : \forall \alpha. (\tau \to \exists \alpha. \tau) \quad \text{unpack}_{\exists \alpha. \tau} : \exists \alpha. \tau \to \forall \beta. ((\forall \alpha. (\tau \to \beta)) \to \beta) \quad (\Delta_{\exists})
$$

These can be read as follows: for any $\alpha$, if you have a $\tau$, then, for some $\alpha$, you have a $\tau$; conversely, if, for some $\alpha$, you have a $\tau$, then, (for any $\beta$) if you wish to obtain a $\beta$ out of $\exists \alpha. \tau$, you must present a function which, for any $\alpha$, obtains a $\beta$ out of a $\tau$. This is somewhat reminiscent of ordinary first-order logic: $\exists x. F$ is equivalent to, and can be defined as, $\neg (\forall x. \neg F)$.

One can go one step further and entirely encode existential types into universal types. This encoding is actually a small example of type-preserving translation! The type translation is double negation:

$$
[\exists \alpha. \tau] = \forall \beta. ((\forall \alpha. ([\tau] \to \beta)) \to \beta) \quad \text{if } \beta \not\equiv \tau
$$

There is actually little choice for the term translation, if the translation is to be type-
preserving:

\[
\begin{align*}
\llbracket \text{pack}_{\exists \alpha. \tau} \rrbracket & : \forall \alpha. (\llbracket \tau \rrbracket \to \llbracket \exists \alpha. \tau \rrbracket) \\
& = \Lambda \alpha. \lambda x : \llbracket \tau \rrbracket. \Lambda \beta. \lambda k : \forall \alpha. (\llbracket \tau \rrbracket \to \beta). k \alpha x \\
\llbracket \text{unpack}_{\exists \alpha. \tau} \rrbracket & : \llbracket \exists \alpha. \tau \rrbracket \to \forall \beta. ((\forall \alpha. (\llbracket \tau \rrbracket \to \beta)) \to \beta) \\
& = \lambda x : \llbracket \exists \alpha. \tau \rrbracket. x
\end{align*}
\]

This encoding is a \textit{continuation-passing transform}. This encoding is due to Reynolds \cite{1983}, although it has more ancient roots in logic.

When existential are presented as constants, their semantics is defined by seeing \texttt{pack}_{\exists \alpha. \tau} as a unary constructor and \texttt{unpack}_{\exists \alpha. \tau} as a unary destructor with the following reduction rule:

\[
\text{unpack}_{\exists \alpha. \tau} \, (\text{pack}_{\exists \alpha. \tau} \, \tau' \, V) \rightarrow \Lambda \beta. \lambda y : \forall \alpha. \tau \to \beta. y \, \tau' \, V \quad (\delta_3)
\]

\textbf{Exercise 42} Show that this \(\delta\)-rule satisfies the progress and subject reduction assumptions for constants with the types in \(\Delta_3\). (You may assume that the standard lemmas still hold.)

(Solution p. 169)

\textbf{Exercise 43} The \(\delta_3\) reduction for existential is permissive it allows reducing of ill-typed terms. Give a more restrictive version of the rule. What will need to be changed in the proof of subject reduction and progress for the \(\delta\)-rule (Exercise 42)?

(Solution p. 169)

Notice that our \(\delta_3\)-reduction reduces an “unpack of a pack” to a polymorphic function that applies its argument to the packed value. This is still a form of continuation-passing-style encoding. It seems more natural to treat \texttt{unpack}_{\exists \alpha. \tau} as a binary destructor to avoid this intermediate step and have the more intuitive reduction rule:

\[
\text{unpack}_{\exists \alpha. \tau} \, (\text{pack}_{\exists \alpha. \tau} \, \tau' \, V) \, \tau_1 \, (\Lambda \alpha. \lambda x : \tau. M) \rightarrow [x \mapsto V][\alpha \mapsto \tau']M \quad (\delta_3)
\]

However, this does not fit in our framework and notion of arity for constants where all type arguments must be passed first and not interleaved with value arguments. Our framework could be extended to the above \(\delta\)-rules for existentials, but the presentation would become cumbersome.

Alternatively, if existential are primitive, their semantics is defined by extending values and evaluation contexts as follows:

\[
V ::= \ldots \ | \ \text{pack} \ \tau', V \ \text{as} \ \tau \quad E ::= \ldots \ | \ \text{pack} \ \tau', [] \ \text{as} \ \tau \ | \ \text{let} \ \alpha, x = \text{unpack} \ [] \ \text{in} \ M
\]

and by adding the following reduction rule:

\[
\text{let} \ \alpha, x = \text{unpack} \ (\text{pack} \ \tau', V \ \text{as} \ \tau) \ \text{in} \ M \rightarrow [\alpha \mapsto \tau'][x \mapsto V]M
\]
6.2. EXISTENTIAL TYPES

Exercise 44  Check that the proofs of subject reduction and progress for System F extend to existential types. (Just check the new cases, assuming that the standard lemmas still hold.) □

The reduction rule for existential destructs its arguments. Hence, let $\alpha, x = \text{unpack} \ M_1$ in $M_2$ cannot be reduced unless $M_1$ is itself a packed expression, which is indeed the case when $M_1$ is a value (or in head normal form). This contrasts with let $x : \tau = M_1$ in $M_2$ where $M_1$ need not be evaluated and may be an application (e.g. in call-by-name or with strong reduction).

Exercise 45  The reduction of let $\alpha, x = \text{unpack} \ M_1$ in $M_2$ could be problematic when $M_1$ is not a value. Illustrate this on an example (You may use the following hint if needed: lanoitidnocasu.)

(Solution p. 169) □

One may wonder whether the pack construct is not too verbose: isn’t the type witness type annotation $\tau'$ in rule PACK superfluous? The type $\tau_0$ of $M$ is fully determined by $M$ and the given type $\exists \alpha. \tau$ of the packed value. Checking that $\tau_0$ is of the form $[\alpha \mapsto \tau'] \tau$ is the matching problem for second-order types, which is simple. However, the reduction rule need the witness type $\tau'$. If it were not available, it would have to be computed during reduction. The reduction rule would then not be pure rewriting. The explicitly-typed language need the witness type for simplicity, while in the surface language, it could be omitted and reconstructed by second-order matching.

6.2.2 Implicitly-type existential types

Intuitively, pack and unpack are just type information that can be dropped by type erasure. More precisely, the erasure of pack $\tau', M$ as $\exists \alpha. \tau \exists \alpha. \tau$ is $M$ and the erasure of let $\alpha, x = \text{unpack} \ M_1$ in $M_2$ is a let-binding let $x = M_1$ in $M_2$. After type-erasure, the following typing rules for existential types in implicit-typed System F:

\[
\begin{align*}
\text{IF-UNPACK} & : \ & \Gamma \vdash a_1 : \exists \alpha. \tau_1 & \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 & \quad \alpha \not\# \tau_2 \\
\Gamma & \vdash \text{let} \ x = a_1 \ \text{in} \ a_2 : \tau_2 \\
\text{IF-PACK} & : \ & \Gamma \vdash a : \alpha \mapsto \tau' \tau & \quad \Gamma \vdash a : \exists \alpha. \tau \\
\end{align*}
\]

Notice, that the let-binding is not typechecked as syntactic sugar for an immediate application. Its semantics remains the same.

\[
E ::= \ldots \text{let} \ x = [] \ \text{in} \ M \quad \text{let} \ x = V \ \text{in} \ M \rightarrow [x \mapsto V]M
\]

Is the semantics still type-erasing? Yes, it is, but there is a subtlety! This is only true in call-by-value. In a call-by-name semantics, a let-bound expression is not reduced prior to substitution of the argument, that is, the rule would be:

let $x = a_1$ in $a_2 \rightarrow [x \mapsto a_1]a_2$

With existential types, this breaks subject reduction!
To see this, let \( \tau_0 \) be \( \exists \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \) and let \( v_0 \) be a value of type \( \text{bool} \). Then, let \( v_1 \) and \( v_2 \) two values of type \( \tau_0 \) with incompatible witness types, taking for instance, \( \lambda f. \lambda x. 1+(f \ (1+x)) \) and \( \lambda f. \lambda x. \text{not} \ (f \ (\text{not} \ x)) \). Let \( v \) be the function \( \lambda b. \text{if} \ b \ \text{then} \ v_1 \ \text{else} \ v_2 \) of type \( \text{bool} \rightarrow \tau_0 \), which returns either one of \( V_1 \) or \( V_2 \) depending on its argument \( b \). We then have the reduction

\[
\begin{align*}
  a_1 &= \text{let } x = v \ v_0 \ \text{in} \ x \ (x \ (\lambda y. y)) \quad \rightarrow \quad v \ v_0 \ (v \ v_0 \ (\lambda y. y)) = a_2 \\
\end{align*}
\]

The typing judgment \( \emptyset \vdash a_1 : \exists \alpha. \alpha \rightarrow \alpha \) holds, while \( \emptyset \vdash a_2 : \tau \) does not hold for any \( \tau \). Indeed, the term \( a_1 \) is well-typed since \( v \ v_0 \) has type \( \tau_0 \), hence \( x \) can be assumed of type \( (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \) for some unknown type \( \beta \) and \( \lambda y. y \) is of type \( \beta \rightarrow \beta \). However, without the outer existential type \( v \ v_0 \) can only be typed with \( (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \exists \alpha. (\alpha \rightarrow \alpha) \), because the value returned by the function need different witnesses for \( \alpha \). This is demanding too much on its argument and the outer application is ill-typed.

One may wonder whether the syntax should not allow the implicit introduction of unpacking instead. For instance, one could argue that if some expression is the expansion of a well-typed let-binding, then it should also be well-typed:

\[
\Gamma \vdash a_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \quad \alpha \neq \tau_2
\]

\[
\Gamma \vdash [x \mapsto a_1] a_2 : \tau_2
\]

However, this rule is not quite satisfactory as it does not have a logical flavor. Moreover, it fixes the previous example, but does not help with the general case: Pick \( a_1 \) that is not yet a value after one reduction step. Then, after let-expansion reduce one of the two occurrences of \( a_1 \). The result is no longer of the form \( [x \mapsto a_1] a_2 \).

In summary, existential types are tricky: The subject reduction property breaks if reduction is not restricted to expressions in head-normal forms. Unrestricted reduction is still safe because well-typedness may eventually be recovered by further reduction steps—so that progress will never break.

Interestingly, the CPS encoding of existential types (1) enforces the evaluation of the packed value (2) before it can be unpacked (3) and substituted (4):

\[
\text{unpack } a_1 \ (\lambda x. a_2) = \llbracket a_1 \rrbracket \ (\lambda x. \llbracket a_2 \rrbracket) \quad (1)
\]

\[
\rightarrow (\lambda k. \llbracket a \rrbracket \ k) \ (\lambda x. \llbracket a_2 \rrbracket) \quad (2)
\]

\[
\rightarrow (\lambda x. \llbracket a_2 \rrbracket) \llbracket a \rrbracket \quad (3)
\]

\[
\rightarrow [x \mapsto \llbracket a_2 \rrbracket] \llbracket a \rrbracket \quad (4)
\]

In the call-by-value setting, \( \lambda k. \llbracket a \rrbracket \ k \) would come from the reduction of \( \text{pack } a \), i.e. is \( (\lambda k. \lambda x. k \ x) \llbracket a \rrbracket \), so that \( a \) is always a value \( v \). However, \( a \) need not be a value. What is essential is again that \( a_1 \) be reduced to some head normal form \( \lambda k. \llbracket a \rrbracket \ k \).
6.2. EXISTENTIAL TYPES

6.2.3 Existential types in ML

What if one wished to extend ML with existential types? Full type inference for existential types is undecidable, just like type inference for universals. However, introducing existential types in ML is easy if one is willing to rely on user-supplied annotations that indicate where to pack and unpack.

This iso-existential approach was suggested by Läufer and Odersky (1994). Iso-existential types are explicitly declared, much as datatypes:

\[ D \bar{\alpha} \approx \exists \bar{\beta}.\tau \quad \text{if ftv}(\tau) \subseteq \bar{\alpha} \cup \bar{\beta} \quad \text{and} \quad \bar{\alpha} \not\approx \bar{\beta} \]

This introduces two constants, with the following type schemes:

\[
\text{pack}_D : \forall \bar{\beta}.\exists \bar{\beta}.\tau \rightarrow D \bar{\alpha} \\
\text{unpack}_D : \forall \bar{\alpha}.\bar{\beta}.D \bar{\alpha} \rightarrow (\forall \bar{\beta}.(\tau \rightarrow \gamma)) \rightarrow \gamma
\]

(Compare with basic iso-recursive types, where \( \bar{\beta} = \emptyset \).)

Unfortunately, the “type scheme” of \( \text{unpack}_D \) is not an ML type scheme. A solution is to make \( \text{unpack}_D \) a binary primitive construct, rather than a constant, with an ad hoc typing rule:

\[
\text{unpack}_D \\
\Gamma \vdash M_1 : D \bar{\alpha} \\
\Gamma \vdash M_2 : \forall \bar{\beta}.([\bar{\alpha} \mapsto \bar{\tau}] \tau \rightarrow \tau_2) \\
\bar{\beta} \not\approx \bar{\tau}, \tau_2 \\
\Gamma \vdash \text{unpack}_D M_1 M_2 : \tau_2
\]

We have seen a version of this rule in System F earlier; this is an ML version. The term \( M_2 \) must be polymorphic, which \text{GEN} can prove.

Iso-existential types are perfectly compatible with ML type inference. The constant \( \text{pack}_D \) admits an ML type scheme, so it is not problematic. The construct \( \text{unpack}_D \) leads to this constraint generation rule (cf. §5):

\[
\langle \text{unpack}_D M_1 M_2 : \tau_2 \rangle = \exists \bar{\alpha}. \left( \langle M_1 : D \bar{\alpha} \rangle \land \forall \bar{\beta}. \langle M_2 : \tau \rightarrow \tau_2 \rangle \right)
\]

where \( D \bar{\alpha} \approx \exists \bar{\beta}.\tau \) and, w.l.o.g., \( \bar{\alpha} \bar{\beta} \not\approx M_1, M_2, \tau_2 \). Note that a universally quantified constraint appears where polymorphism is required.

In practice, Läufer and Odersky suggest fusing iso-existential types with algebraic data types. The somewhat bizarre Haskell syntax for this is:

\[
data D \bar{\alpha} = \forall \bar{\beta}.\ell \tau
\]

where \( \ell \) is a data constructor. The elimination construct \( \langle \text{case} M_1 \text{ of } \ell \ x \rightarrow M_2 : \tau_2 \rangle \) and is typed as follows:

\[
\langle \text{case} M_1 \text{ of } \ell \ x \rightarrow M_2 : \tau_2 \rangle = \exists \bar{\alpha}. \left( \langle M_1 : D \bar{\alpha} \rangle \land \forall \bar{\beta}.\text{def } x : \tau \text{ in } \langle M_2 : \tau_2 \rangle \right)
\]

where, w.l.o.g., \( \bar{\alpha} \bar{\beta} \not\approx M_1, M_2, \tau_2 \).
Examples Define $\text{Any} \approx \exists \beta. \beta$. The following code that attempts to extract the raw content of a package fails:

$$\langle \text{unpack}_{\text{Any}} M_1 (\lambda x. x : \tau_2) \rangle = \langle M_1 : \text{Any} \rangle \land \forall \beta. \langle \lambda x. x : \beta \rightarrow \tau_2 \rangle \equiv \forall \beta. \beta = \tau_2 \equiv \text{false}$$

Now, define $D \alpha \approx \exists \beta. (\beta \rightarrow \alpha) \times \beta$. A client that regards $\beta$ as abstract succeeds:

$$\langle \text{unpack} D M_1 (\lambda (f, y). f y : \tau) \rangle = \exists \alpha. (\langle M_1 : D \alpha \rangle \land \forall \beta. \langle \lambda (f, y). f y : ((\beta \rightarrow \alpha) \times \beta) \rightarrow \tau \rangle)$$

Remark 6 We reuse the type $D \alpha \approx \exists \beta. (\beta \rightarrow \alpha) \times \beta$ of frozen computations, defined above. Assume given a list $l$ of elements of type $D \tau_1$. Assume given a function $g$ of type $\tau_1 \rightarrow \tau_2$. We may transform the list into a new list $l'$ of frozen computations of type $D \tau_2$ (without actually running any computation).

$$\text{List.map} (\lambda (z) \text{ let } D (f, y) = z \text{ in } D ((\lambda (z) g (f z)), y))$$

We may generalize the code into a functional that receives $g$ and $l$ as arguments and returns $l'$. Unfortunately, the following code does not typecheck:

$$\text{let lift } g \ l = \text{List.map} (\lambda (z) \text{ let } D (f, y) = z \text{ in } D ((\lambda (z) g (f z)), y))$$

The problem is that, in expression $\text{let } \alpha, x = \text{unpack} M_1$ in $M_2$, occurrences of $x$ can only be passed to polymorphic functions so that the type $\alpha$ of $x$ does not escape from its scope. That is first-class existential types fail for first-class universal types as well!

Mitchell and Plotkin (1988) note that existential types offer a means of explaining abstract types. For instance, the type:

$$\exists \text{stack.\{} \text{ empty : stack; push : int } \times \text{ stack } \rightarrow \text{ stack; pop : stack } \rightarrow \text{ option } (\text{int } \times \text{ stack}) \}\$$

specifies an abstract implementation of integer stacks.

Unfortunately, it was soon noticed that the elimination rule is too awkward, and that existential types alone do not allow designing module systems Harper and Pierce (2005), Montagu and Rémy (2009) make existential types more flexible in several important ways, and argue that they might explain modules after all.

6.2.4 Existential types in OCaml

Amusingly, existential types were first available in OCaml via abstract types and first-class modules. There are now also available as a degenerate case of Generalized Algebraic DataTypes (GADT) which coincides with the approached described above.

For example, one may defined the previous datatype of frozen computations:
6.3. **Typed closure conversion**

Equipped with existential types, we may now revisit type closure conversion.

### 6.3.1 Environment-passing closure conversion

Remember that we came to the conclusion that the translation of arrow types $[\tau_1 \to \tau_2]$ must be $\exists \alpha.((\alpha \times [\tau_1]) \to [\tau_2]) \times \alpha$. Let us show that we may translate expressions so as to preserve well-typedness, i.e. so that $\Gamma \vdash M : \tau$ implies $[\Gamma] \vdash [M] : [\tau]$. Assume $\Gamma \vdash \lambda x. M : \tau_1 \to \tau_2$ and $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\lambda x. M)$. We may now hide the dependence on $\Gamma$ using an existential type:

$$
[\lambda x : \tau_1. M] = \text{let code : } ([\Gamma] \times [\tau_1]) \to [\tau_2] = \\
\lambda (env : [\Gamma], x : [\tau_1]). \text{let } (x_1, \ldots, x_n : [\Gamma]) = env \text{ in } [M] \text{ in } \\
\text{pack } [\Gamma], (code, (x_1, \ldots, x_n)) \text{ as } \exists \alpha.((\alpha \times [\tau_1]) \to [\tau_2]) \times \alpha \\
: \exists \alpha.((\alpha \times [\tau_1]) \to [\tau_2]) \times \alpha = [\tau_1 \to \tau_2]
$$

In the case of application, assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash M_1 : \tau_1$ and take:

$$
[M M_1] = \text{let } \alpha, (code : (\alpha \times [\tau_1]) \to [\tau_2], env : \alpha) = \text{unpack } [M] \text{ in } code (env, [M_1]) \\
: [\tau_2]
$$

For recursive functions we may use the “fix-code” variant (Morrisett and Harper, 1998):

$$
[\mu f . \lambda x. M] = \text{let rec code (env, x) = } \\
\text{let } f = \text{pack } (code, env) \text{ in let } (x_1, \ldots, x_n) = env \text{ in } [M] \text{ in } \\
\text{pack } (code, (x_1, \ldots, x_n))
$$

where $\{x_1, \ldots, x_n\} = \text{fv}(\mu f . \lambda x. M)$. The translation of applications is unchanged as recursive and non-recursive functions have an identical calling convention. This translation builds recursive code, avoiding a recursive closure, hence the code is easy to type. Unfortunately, as a counterpart, a new closure is allocated at every call, which is the weak point of this variant.
Instead, the “fix-pack” variant (Morrisett and Harper, 1998) uses an extra field in the environment to store a back pointer to the closure:

\[
\mu f. \lambda x. M = \text{let code} = \lambda (env, x). \text{let } (f, x_1, \ldots, x_n) = \text{env in } [M] \text{ in}
\]

\[
\text{let rec } clo = (\text{code}, (clo, x_1, \ldots, x_n)) \text{ in } clo
\]

where \(\{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M)\). Hence, we avoid rebuilding the closure at every call by creating a recursive closure. However, this requires, in general, recursively-defined values and closures are now cyclic data structures.

Here is how the “fix-pack” variant is type-checked. Assume \(\Gamma \vdash \mu f. \lambda x. M : \tau_1 \rightarrow \tau_2\) and \(\text{dom}(\Gamma) = \{x_1, \ldots, x_n\} = \text{fv}(\mu f. \lambda x. M)\).

\[
[\mu f : \tau_1 \rightarrow \tau_2, \lambda x. M] =
\]

\[
\text{let code} : \left(\left[ f : \tau_1 \rightarrow \tau_2; \Gamma \right] \times [\tau_1]\right) \rightarrow [\tau_2] =
\]

\[
\lambda (env : \left[ f : \tau_1 \rightarrow \tau_2, \Gamma \right], x : [\tau_1]). \text{let } (f, x_1, \ldots, x_n) : \left[ f : \tau_1 \rightarrow \tau_2, \Gamma \right] = \text{env in } [M] \text{ in}
\]

\[
\text{let rec } clo : \left[ \tau_1 \rightarrow \tau_2 \right] =
\]

\[
\text{pack } \left[ f : \tau_1 \rightarrow \tau_2, \Gamma \right], (\text{code}, (clo, x_1, \ldots, x_n)) \text{ as } \exists \alpha ((\alpha \times [\tau_1]) \rightarrow [\tau_2]) \times \alpha
\]

in \(clo\)

This implements monomorphic recursion, as by default in ML. To allow the recursive function to be polymorphic, we can generalize the encoding afterwards:

\[
[\Lambda \vec{\beta}, \mu f : \tau_1 \rightarrow \tau_2, \lambda x. M] = \Lambda \vec{\beta}. [\mu f : \tau_1 \rightarrow \tau_2, \lambda x. M]
\]

whenever the right-hand side is well-defined. This allows the polymorphic functions to be polymorphic functions as long as the recursion is monomorphic.

Fortunately, the encoding can be straightforwardly adapted to directly compile polymorphically recursive functions into polymorphic closure.

\[
[\mu f : \forall \vec{\beta}. \tau_1 \rightarrow \tau_2, \lambda x. M] =
\]

\[
\text{let code} : \forall \vec{\beta}. \left(\left[ f : \forall \vec{\beta}. \tau_1 \rightarrow \tau_2; \Gamma \right] \times [\tau_1]\right) \rightarrow [\tau_2] =
\]

\[
\Lambda \vec{\beta}. \lambda (env : \left[ f : \forall \vec{\beta}. \tau_1 \rightarrow \tau_2, \Gamma \right], x : [\tau_1]).
\]

\[
\text{let } (f, x_1, \ldots, x_n) : \left[ f : \forall \vec{\beta}. \tau_1 \rightarrow \tau_2, \Gamma \right] = \text{env in } [M] \text{ in}
\]

\[
\text{let rec } clo : \left[ \forall \vec{\beta}, \tau_1 \rightarrow \tau_2 \right] = \Lambda \vec{\beta}.
\]

\[
\text{pack } \left[ f : \forall \vec{\beta}, \tau_1 \rightarrow \tau_2, \Gamma \right], (\text{code } \vec{\beta}, (clo, x_1, \ldots, x_n)) \text{ as } \exists \alpha ((\alpha \times [\tau_1]) \rightarrow [\tau_2]) \times \alpha
\]

in \(clo\)

In summary, the environment-passing closure conversion is simple, but it requires the introduction of recursive non-functional values \(\text{let rec } x = V \text{ in } M\). While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof (as recursive non-functional values were not permitted so far).
6.3. TYPED CLOSURE CONVERSION

6.3.2 Closure-passing closure conversion

Recall the closure-passing variant:

\[
\lambda x. M = \text{let } \text{code} = \lambda (\text{clo}, x). \text{let } (\_ , x_1 , \ldots , x_n ) = \text{clo} \text{ in } \lbrack M \rbrack \text{ in (code, } x_1 , \ldots , x_n )
\]

\[
\lbrack M_1 \ M_2 \rbrack = \text{let } \text{clo} = \lbrack M_1 \rbrack \text{ in let } \text{code} = \text{proj}_0 \text{ clo} \text{ in code (clo, } \lbrack M_2 \rbrack)\]

where \( \{x_1, \ldots , x_n \} = \text{fv}(\lambda x. M) \).

There are two difficulties to typecheck this: first, a closure is a tuple, whose first field—the code pointer—should be exposed, while the number and types of the remaining fields—the environment—should be abstract; second, the first field of the closure contains a function that expects the closure itself as its first argument.

To describe this, we use two type-theoretic mechanisms; first existential quantification over the tail of a tuple (a.k.a. a row) to allow the environment to remain abstract; and recursive types to allow the closure to point to itself.

Tuples, rows, row variables  Let us first introduce extensible tuples. The standard tuple types that we have used so far are:

\[
\tau ::= \ldots | \Pi R \quad \text{— types}
\]

\[
R ::= \epsilon | (\tau; R) \quad \text{— rows}
\]

The notation \((\tau_1 \times \ldots \times \tau_n)\) was sugar for \(\Pi (\tau_1; \ldots ; \tau_n; \epsilon)\). Let us introduce row variables and allow quantification over them:

\[
\tau ::= \ldots | \Pi R | \forall \rho. \tau | \exists \rho. \tau \quad \text{— types}
\]

\[
R ::= \rho | \epsilon | (\tau; R) \quad \text{— rows}
\]

This allows reasoning about the first few fields of a tuple whose length is not known. The typing rules for tuple construction and deconstruction are:

\[
\text{TUPLE} \quad \forall i. \epsilon [1, n] \quad \Gamma \vdash M_i : \tau_i \quad \frac{\Gamma \vdash (M_1, \ldots , M_n) : \Pi (\tau_1; \ldots ; \tau_n; \epsilon)}{\Gamma \vdash \text{proj}_i M : \tau_i}
\]

These rules make sense with or without row variables. Projection does not care about the fields beyond \( i \). Thanks to row variables, this can be expressed in terms of parametric polymorphism: \( \text{proj}_i : \forall \alpha_1 \ldots \alpha_i \rho. \Pi (\alpha_1 ; \ldots ; \alpha_i ; \rho) \rightarrow \alpha_i \).

Remark 7 Rows were invented by Wand (1988) and improved by Rémy (1994b) in order to ascribe precise types to operations on records. The case of tuples, presented here, is simpler. Rows are used to describe objects in OCaml (Rémy and Vouillon, 1998). Rows are explained in depth by Pottier and Rémy (2005).
Back to closure-passing closure conversion. Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$[\tau_1 \to \tau_2] = \exists \rho. \mu \alpha. \Pi (((\alpha \times [\tau_1]) \to [\tau_2]); \rho)$$

$\rho$ describes the environment represented as a row of fields, which is abstract; $\alpha$ is the concrete type of the closure that is to refer to recursively; $\Pi (((\alpha \times [\tau_1]) \to [\tau_2]); \rho)$ is a tuple that begins with a code pointer of type $(\alpha \times [\tau_1]) \to [\tau_2]$ and continues with the environment $\rho$. Notice that $\tau$ appears only once, which may seem surprising. Usually, an existential type variable appears both at positive and negative occurrences. Here, the variable $\alpha$ appears only at a negative occurrence, but in a recursive part of the type that can be unfolded.

To help checking well-typedness of the encoding, let $\text{Clo}(R)$ abbreviate the concrete type of a closure of row $R$ and $\text{UClo}(R)$ its unfolded version:

$$\text{Clo}(R) \triangleq \mu \alpha. \Pi (((\alpha \times [\tau_1]) \to [\tau_2]; \rho)$$

$$\text{UClo}(R) \triangleq \Pi (((\text{Clo}(R) \times [\tau_1]) \to [\tau_2]; \rho)$$

The encoding of arrow types $[\tau_1 \to \tau_2]$ is $\exists \rho. \text{Clo}(\rho)$. The encoding of abstractions and applications is:

$$[[\lambda x: \tau_1. M]] = \text{let code : (\text{Clo}([\Gamma]) \times [\tau_1]) \to [\tau_2]} =$$

$$\lambda (clo : \text{Clo}([\Gamma]), x : [\tau_1]).$$

$$\text{let } (\_ : x_1, \ldots , x_n) : \text{UClo}([\Gamma]) = \text{unfold } clo \text{ in } [M] \text{ in }$$

$$\text{pack } [\Gamma], (\text{fold } (\text{code}, x_1, \ldots , x_n)) \text{ as } \exists \rho. \text{Clo}(\rho)$$

$$[[M_1 M_2]] = \text{let } \rho, clo = \text{unpack } [M_1] \text{ in }$$

$$\text{let code : (\text{Clo}(\rho) \times [\tau_1]) \to [\tau_2]} = \text{proj}_0 (\text{unfold } clo) \text{ in }$$

$$\text{code } (clo, [M_2])$$

where $\{x_1, \ldots , x_n\} = \text{fv}(\lambda x. M)$.

In the closure-passing variant, recursive functions can be translated as follows:

$$[[\mu f. \lambda x. M]] = \text{let code } = \lambda (clo, x).$$

$$\text{let } f = clo \text{ in let } (\_ : x_1, \ldots , x_n) = clo \text{ in } [M] \text{ in }$$

$$(\text{code}, x_1, \ldots , x_n)$$

where $\{x_1, \ldots , x_n\} = \text{fv}(\mu f. \lambda x. M)$. No extra field or extra work is required to store or construct a representation of the free variable $f$: the closure itself plays this role. However, this untyped code can only be typechecked when recursion is monomorphic.

Exercise 46. Carefully check well-typedness of the above translation with monomorphic recursion. \(\square\)
6.3. TYPED CLOSURE CONVERSION

To adapt this encoding to polymorphic recursion, the problem is that recursive occurrences of $f$ are rebuilt from the current invocation of the closure, this with the same type since the closure is invoked after type specialization.

By contrast, in the environment passing encoding, the environment contained a polymorphic binding for the recursive calls that was filled with the closure before its invocation, i.e. with a polymorphic type.

Fortunately, we may slightly change the encoding, using a recursive closure as in the type-passing version, to allow type checking in System $F$.

**Remark 8** One could think of changing the encoding of closure types $\lceil \tau_1 \to \tau_2 \rceil$ to make the encoding work. However, although this should be possible in some more expressive type systems, there seems to be no easy way to do so and certainly not within System $F$.

Let $\tau$ be $\forall \vec{\alpha}. \tau_1 \to \tau_2$ and $\Gamma_f$ be $f : \tau, \Gamma$ where $\vec{\beta} \neq \Gamma$.

\[
\lceil \mu f : \tau. \lambda x. M \rceil = \\text{let code} = \\
\text{let } (\text{code, } f, x_1, \ldots, x_n) : \forall \vec{\beta}. \text{UClo}([\Gamma_f]) = \text{unfold } \text{clo} \text{ in } [M] \text{ in} \\
\text{let rec } \text{clo} : \forall \vec{\beta}. \exists \rho. \text{Clo}(\rho) = \\
\text{let } \text{clo} \text{ in } [\Gamma], (\text{fold } (\text{code } \vec{\beta}, \text{clo}, x_1, \ldots, x_n)) \text{ as } \exists \rho. \text{Clo}(\rho)
\]

Remind that $\text{Clo}(R)$ abbreviates $\mu \alpha. \Pi ((\alpha \times [\tau_1]) \to [\tau_2]; R)$. Hence, $\vec{\beta}$ are free variables of $\text{Clo}(R)$. Here, a polymorphic recursive function is directly compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged, so the encoding of applications is also unchanged.

**Optimizing representations** Closure-passing and environment-passing closure conversions cannot be mixed because the calling-convention (i.e., the encoding of application) must be uniform. However, there is some flexibility in the representation of the closure. For instance, the following change is completely local:

\[
\lceil \lambda x. M \rceil = \text{let code} = \lambda (\text{clo, } x). \text{let } (\text{code, } (x_1, \ldots, x_n)) = \text{clo} \text{ in } [M] \text{ in} \\
(code, (x_1, \ldots, x_n))
\]

This allows for sharing the closure (or part of it) may be shared when many definitions share the same closure.

6.3.3 Mutually recursive functions

Can we compile mutually recursive functions $\mu(f_1, f_2). (\lambda x_1. M_1, \lambda x_2. M_2)$, say $M$?
CHAPTER 6. EXISTENTIAL TYPES

The environment passing encoding is as follows:

\[
[M] = \text{let } \text{code}_i = \lambda (env, x). \text{let } (f_1, f_2, x_1, \ldots, x_n) = env \text{ in } [M_i] \text{ in }
\]

\[
\text{let rec } env = (clo_1, clo_2, x_1, \ldots, x_n)
\]

\[
\text{and } clo_1 = (\text{code}_1, env)
\]

\[
\text{and } clo_2 = (\text{code}_2, env) \text{ in }
\]

clo_1, clo_2

Notice that we can share the environment inside the two closures. The closure passing encoding is:

\[
[M] = \text{let } \text{code}_i = \lambda (clo, x). \text{let } (\_ , f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_i] \text{ in }
\]

\[
\text{let rec } clo = (\text{code}_1, clo_1, clo_2, x_1, \ldots, x_n)
\]

\[
\text{and } clo_2 = (\text{code}_2, clo_1, clo_2, x_1, \ldots, x_n) \text{ in } clo_1, clo_2
\]

Question: Can we share the closures c_1 and c_2 in case n is large?

Here the environment cannot be shared between the two closures, since they belong to tuples of different size. Unless the runtime, in particular the garbage collector, supports such an operation as returning the tail of a tuple without allocating a new tuple. Then we could write:

\[
[M] = \text{let } \text{code}_1 = \lambda (clo, x). \text{let } (\_ , f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_i] \text{ in }
\]

\[
\text{let } \text{code}_2 = \lambda (clo, x). \text{let } (\_ , f_1, f_2, x_1, \ldots, x_n) = clo \text{ in } [M_j] \text{ in }
\]

\[
\text{let rec } clo_1 = (\text{code}_1, \text{code}_2, clo_1, clo_2, x_1, \ldots, x_n)
\]

\[
\text{and } clo_2 = clo_1.\text{tail} \text{ in } clo_1, clo_2
\]

Here clo_1.\text{tail} returns a pointer to the tail (\text{code}_2, clo_1, clo_2, x_1, \ldots, x_n) of clo_1 without allocating a new tuple.

**Encoding of objects** The closure-passing representation of mutually recursive functions is similar to the representation of objects in the object-as-record-of-functions paradigm:

A class definition is an object generator:

```
class c (x_1, \ldots x_q) \{ \text{meth } m_1 = M_i; \ldots \text{meth } m_q = M_i \}
```

Given arguments for parameter x_1, \ldots x_n, it builds recursive methods m_1, \ldots m_n. A class can be compiled into an object closure:

```
let m =
\{ 
   m_1 = \lambda (m, x_1, \ldots x_q). [M_i];
   \vdots
   m_p = \lambda (m, x_1, \ldots x_q). [M_p] \} \text{ in }
\lambda x_1, \ldots x_q. (m, x_1, \ldots x_q)
```

Each m_i is bound to the code for the corresponding method. All codes are combined into a
record of codes. Then, calling method \( m_i \) of an object \( p \) is \((\text{proj}_0 p).m_ip\).

Let us write the typed version of this encoding. Let \( \tau_i \) be the type of \( M_i \) and row \( R \) describe the types of \((x_1,\ldots,x_q)\). Let \( \text{Clo}(R) \) be \( \mu\alpha.\Pi((m_i: \alpha \rightarrow \tau_i)^{i\in I}; R) \) and \( \text{UClo}(R) \) its unfolding.

Fields \( R \) are hidden in an existential type \( \mu\alpha.\Pi((m_i: \alpha \rightarrow \tau_i)^{i\in I}; \rho) \):

\[
\begin{align*}
\text{let } m &= \{ \\
&= \lambda(m,x_1,\ldots,x_q: \text{UClo}(R)).[[M_1]]; \\
&\vdots \\
&\lambda x_1.\ldots\lambda x_q. \text{pack } R.\text{fold} (m,x_1,\ldots,x_q) \text{ as } \exists \rho.([M_1],\rho) \\
\}
\end{align*}
\]

Calling a method of an object \( p \) of type \( M \) is

\[
\begin{align*}
p#m_i & \triangleq \text{let } \rho, z = \text{unpack} p \text{ in } (\text{proj}_0 \text{ unfolding } z).m_iz
\end{align*}
\]

An object has a recursive type but it is not a recursive value.

Typed encoding of objects were first studied in the 90’s to understand what objects really are in a type setting. These encodings are in fact type-preserving compilation of (primitive) objects. There are several variations on these encodings. See Bruce et al. (1999) for a comparison. See Remy (1994a) for an encoding of objects in (a small extension of) ML with iso-existentials and universals. See Abadi and Cardelli (1996, 1995) for more details on primitive objects.

### Summary

Type-preserving compilation is rather fun. (Yes, really!) It forces compiler writers to make the structure of the compiled program fully explicit, in type-theoretic terms. In practice, building explicit type derivations, ensuring that they remain small and can be efficiently typechecked, can be a lot of work.

Because we have focused on type preservation, we have studied only naive closure conversion algorithms. More ambitious versions of closure conversion require program analysis: see, for instance, Steckler and Wand (1997). These versions can be made type-preserving.

Defunctionalization, an alternative to closure conversion, offers an interesting challenge, with a simple solution. See, for instance, Pottier and Gauthier (2006). Designing an efficient, type-preserving compiler for an object-oriented language is quite challenging. See, for instance, Chen and Tarditi (2005).

One may think that references in System F could be translated away by making the store explicit. In fact, this can be done, but not in System F, nor even in System F\( ^\omega \): the translation is quite tricky and in order for the translation to be well-typed the type system must be reach enough to express monotonicity of the store in a context where the store is itself recursively defined. See Pottier (2011) for details.
Exercise 47 (CPS conversion) Here is an untyped version of call-by-value CPS conversion:

\[
\begin{align*}
\llbracket V \rrbracket &= \lambda k. k (\llbracket V \rrbracket) \\
\llbracket M_1 M_2 \rrbracket &= \lambda k. \llbracket M_1 \rrbracket (\lambda x_1. \llbracket M_2 \rrbracket (\lambda x_2. x_1 x_2 k))
\end{align*}
\]

\[
\begin{align*}
\llbracket x \rrbracket &= x \\
\llbracket () \rrbracket &= () \\
\llbracket (V_1, V_2) \rrbracket &= (\llbracket V_1 \rrbracket, \llbracket V_2 \rrbracket) \\
\llbracket \lambda x. M \rrbracket &= \lambda x. \llbracket M \rrbracket
\end{align*}
\]

Is this a type-preserving transformation? (Solution p. 169)
Chapter 7

Overloading

7.1 An overview

Overloading occurs when several definitions of an identifier may be visible simultaneously at the same occurrence in a program. An interpretation of the program (and a fortiori a run of the program) must choose the definition that applies at this occurrence. This is called overloading resolution. Overloading resolution may use quite different strategies and techniques. All sorts of identifiers may be subject to overloading: variables, labels, constructors, types, etc.

Overloading must be distinguished from shadowing of identifiers by normal scoping rules, where in this case, a definition is just temporarily inaccessible by another one, but only the last definition is visible.

7.1.1 Why use overloading?

There are several reasons to use overloading.

Overloading may just be a naming convenience that allows reusing the same identifier for similar but different operations. This avoids name mangling such as suffixing similar names by type information: printing functions, e.g. `print_int`, `print_string`, etc.; numerical operations, e.g. `(+)`, `.+` etc.; or numerical constants e.g. `0`, `.0`, etc. In this respect, it may help with modularity. In the absence of overloading, the naming discipline (including name mangling conventions) must be known globally to avoid name clashes, which breaks compositionality. Isolated identifiers with no particular naming convention may still interfere between different developments and cannot be used together unless fully qualified. This problem does not disappear with overloading but it may be minimized—as long as overloading is not ambiguous. Hence, in some sense, overloading allows to think more abstractly, in terms of operations rather than of particular implementations. For instance, calling `to_string` conversion lets the system check whether one definition is available according to the type of the argument.
Overloading definitions may also be used to provide type dependent functions. That is, a function may be defined for all types $\tau[\alpha]$ but with an implementation depending on the type of $\alpha$ by providing several overloaded definitions for different types $\tau[\tau_i]$. For instance, a marshaling function of type $\forall \alpha. \alpha \rightarrow \text{string}$ may execute different code for each base type $\alpha$.

Overloading definitions may be ad hoc, i.e. completely unrelated for each type—or just share a same type schema. For example, $0$ could mean either the integer zero or the empty list; and “$\times$” could mean either the integer product or string concatenation.

Conversely, overloaded definitions may depend solely on the type structure (i.e. on whether the argument is a sum, a product, etc.) so that definitions can be derived mechanically for all types from their definitions on base types. Such overloaded functions are called polytypic functions. Typical examples are marshaling functions, or the generation of random values for arbitrary types as used in the Quickcheck tool for Haskell, etc. Still, polytypic definition often need to be specialize at some particular types. For example, one may use a polytypical definition of printing, so that printing is available at all types, but define specialized versions of printing at some particular types.

### 7.1.2 Different forms of overloading

There are many variants of overloading. They can be classified by how overloading is introduced and resolved.

The first elements of classification are the restrictions on overloading definitions. Can arbitrary definitions be overloaded? For instance, can numerical values be overloaded? Are all overloaded definitions of the same symbol instances of a common type scheme? Are these type schemes arbitrary? Are overloaded definitions primitive (pre-existing), automatic (generated mechanically from other definitions), or user-defined? Can overloaded definitions overlap? Can overloaded definitions have a local scope?

However, the main element of classification remains the resolution strategy—which may indirectly constraint the way overloading is introduced. We distinguish between static and dynamic resolutions strategies.

Static resolution of overloading has a very simple semantics since the meaning of the program can be determined statically by deciding for each overloaded symbol which actual definition of the symbol should be used. Hence, it replaces each occurrence of an overloaded symbol by an actual implementation at the appropriate type. Therefore static overloading does not increase expressiveness per say, since the user could have chosen the appropriate implementation in the first place. Still, static overloading may significantly reduce verbosity—and increase modularity and abstraction, as explained above.

Conversely, dynamic resolution increases expressiveness, as the choice of the implementation may now depend on the dynamic of the program execution. However, it is also much more involved, since the semantics of the language usually need extra machinery to support the dynamic resolution. For example, the resolution of some occurrence of a polymorphic
function may depend on the type of its arguments, so that different calls of the function
at different types can make different choices. The resolution is driven by information made
available at runtime: it could at worse require full type information. In some restrictions,
partial type information may be sufficient, and sometimes some type-related information can
be used instead of types themselves, such as tags, dictionaries, etc. These can be attached
to values (as tags in object oriented languages), or passed as extra arguments at runtime (as
dictionaries in Haskell).

7.1.3 Static overloading

The language SML has a very limited form of overloading where overloaded definitions are
primitive: they include an exhaustive list of overloaded definitions for numerical operators,
plus automatically generated overloaded definitions for all record accessors. The resolution is
static and fails if overloading cannot be unambiguously resolved at outermost let-definitions.
For example, \texttt{let twice x = x + x} is rejected in SML at toplevel, since + could be either the
addition on either integers or floats.

In the language Java, overloading is not primitive but automatically generated by sub-
typing: when a class extends another one and a method is redefined, the older definition
is still visible, but at another type, hence the method is overloaded. This overloading is
then statically resolved by choosing the most specific definition. There is always a best
choice—according to static knowledge. This static resolution of overloading in Java comes
in complement to the dynamic dispatch of method calls. This is often a source of confusion
for programmers who often expect a dynamic resolution of overloading and as a result mis-
understand the semantics of their programs. For instance, an argument may have a runtime
type that is a subtype of the best known compile-time type, and perhaps a more specific
definition could have been used if overloading were resolved dynamically.

However convenient, static resolution of overloading is quite limited. Moreover, it does
not fit very well with first-class functions and polymorphism. Indeed, with static overloading,
\texttt{\lambda x. x + x} is rejected when + is overloaded, as it cannot be resolved. The function must be
manually specialized at some type for which + is defined. This argues in favor of some form
of dynamic overloading that allows to delay resolution of overloaded symbols at least until
polymorphic functions have been sufficiently specialized.

7.1.4 Dynamic resolution with a type passing semantics

The most ambitious approach to dynamic overloading is to pass types at runtime and dis-
patch on the runtime type, using a general typecase construct.

Runtime type dispatch is the most general approach as it does not impose much restric-
tion on the introduction of overloaded definitions. It uses an explicitly-typed calculus (\textit{e.g.}
System F)—with a type passing semantics—extended with a typecase construct. However,
the runtime cost of typecase may be high, unless type patterns are significantly restricted. Moreover, one pays even when overloading is not used, since types are always passed around, even when overloading is not used, unless the compiler uses aggressive program analyzes to detect these situations and optimize type computations away. Monomorphization may also be used to allow more static resolution in such cases. Ensuring exhaustiveness of type matching is often a difficult task in this context.

The ML& calculus by Castagna (1997) offers a general overloading mechanism based on type dispatch. It is an extension of System F with intersection types, subtyping, and type matching. An expressive type system keeps track of exhaustiveness; type matching functions are first-class and can be extended or overridden. The language allows overlapping definitions with a best match resolution strategy.

### 7.1.5 Dynamic overloading with a type erasing semantics

To avoid the expensive cost of typecase, one may restrict the overloaded definitions, so that full type information is not needed and only an approximation of types, such as tags, may be used for overloading resolution. This is one possible approach to object-orientation in the method as overloading functions paradigm where object classes are used to dynamically select the appropriate method. This is also an approach used in some scheme dialects known as generics.

In fact, one may get more freedom by detaching tags from values and passing tags—or almost equivalently passing the actually implementations grouped into dictionaries—as extra runtime arguments. A side advantage of this approach is that the semantics can be described without changing the runtime environment, i.e. the representation of values, as an elaboration process that introduces abstractions and applications for implementations of overloaded symbols. Schematically, one transforms unresolved overloaded symbols into extra abstractions and passes actual implementations (or abstractions of implementations) around as extra arguments. Hopefully, overloaded symbols can be resolved when their types are sufficiently specialized and before they are actually needed.

For example, a program context \( \text{let } f = \lambda x. x + x \text{ in } [\ ] \) can be elaborated into \( \text{let } f = \lambda (+). \lambda x. x + x \text{ in } [\ ] \). If \( f 1.0 \) is placed in the hole of this original program context, it can then be elaborated to \( f (+. \) 1.0, which can be placed in the hole of the elaborated program context. Elaboration can be performed after typechecking by translating the typing derivation. After elaboration, types are no longer needed and can be erased. Monomorphization or other simplifications may reduce the number of abstractions and applications introduced by overloading resolution.

This technique has been widely explored—under different facets—in the context of ML: Type classes, introduced very early by Wadler and Blott (1989) are still the most popular and widely used framework. Other contemporary solutions have been proposed by Rouaix (1990) and Kaes (1992). Simplifications of type classes have also been proposed by Odersky et al.
In the rest of this chapter we introduce a tiny language called Mini Haskell that models the essence of Haskell type classes; at the end we also discuss implicit arguments as a less structured but simpler way of introducing dynamic overloading in a programming language.

7.2 Mini Haskell

Mini Haskell—or MH for short—is a simplification of Haskell to avoid most of the difficulties of type classes but keeping their essence: it is restricted to single parameter type classes and no overlapping instance definitions; it is close in expressiveness and simplicity to A second look at overloading but closer to Haskell in style—it can be easily generalized by lifting restrictions without changing the framework.

The language MH is explicitly typed. In this section, we first present some examples in MH, and then describe the language and its elaboration into System F. We introduce an implicitly-typed version of MH and its elaboration in the next section.

7.2.1 Examples in MH

An equality class and several instances many be defined in Mini Haskell as follows:

```haskell
class Eq (X)  { equal : X -> X -> Bool }
inst Eq (Int) { equal = (==) }
inst Eq (Char) { equal = (==) }
inst Λ(X) Eq (X) ⇒ Eq (List (X))
  { equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
    | [],[] → true | [],_ | _,[] → false
    | h1::t1, h2::t2 → equal X h1 h2 && equal (List X) t1 t2 }
```

This code declares a class (dictionary) of type Eq(X) that contains definitions for equal : X -> X -> X and creates two concrete instances (dictionaries) of type Eq(Int) and Eq(Char), and a function that, given a dictionary for Eq(X), builds a dictionary for type List(X). This code can be elaborated by explicitly building dictionaries as records of functions:

```haskell
type Eq (X) = { equal : X -> X -> Bool }
let equal X (EqX : Eq X) : X -> X -> Bool = EqX.equal

let EqInt : Eq Int = { equal = ( (==) : Int -> Int -> Bool ) }
let EqChar : Eq Char = { equal = primEqChar }

let EqList X (EqX : Eq X) : Eq (List X) =
  { equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
    | [],[] → true | [],_ | _,[] → false
    | h1::t1, h2::t2 → 
```

Classes may themselves depend on other classes (called superclasses), which realizes a form of class inheritance.

```haskell
class Eq (X) ⇒ Ord (X) { lt : X → X → Bool }
inst Ord (Int) { lt = (<=) }
```

The class definition declares a new class (dictionary) `Ord (X)` that contains a method `Ord (X)` that depends on a dictionary `Eq (X)` and contains a method `lt : X → X → Bool`. The instance definition builds a dictionary `Ord (Int)` from the existing dictionary `Eq Int` and the primitive `(<=)` for `lt`. The two declarations are elaborated into:

```haskell
type Ord (X) = { Eq : Eq (X); lt : X → X → Bool }
let EqOrd X (OrdX : Ord X) : Eq X = OrdX.Eq
let lt X (OrdX : Ord X) : X → X → Bool = OrdX.lt
let OrdInt : Ord Int = { Eq = EqInt; lt = (<=) }
```

So far, we have just defined type classes and some instances. We may write a function that uses these overloaded definitions. When overloading cannot be resolved statically, the function will be abstracted other one or several additional arguments, called dictionaries, that will carry the appropriate definitions for the unresolved overloaded symbols. For example, consider the following definition in Mini Haskell:

```haskell
let rec search : ∀(X) Ord (X) ⇒ X → List X → Bool =
Λ(X) λ(x : X) λ(l : List X)
match l with [] → false | h::t → equal X (EqOrd X OrdX) x h || search X x t
```

This code is elaborated into:

```haskell
let rec search X (OrdX : Ord X) (x : X) (l : List X) : Bool =
match l with [] → false
| h::t → equal X (EqOrd X OrdX) x h || search X OrdX x t
```

Using the overloading function, as in `search Int 1 [1; 2; 3]` will then elaborate into the code `search Int OrdInt 1 [1; 2; 3]` where a dictionary `OrdInt` of the appropriate type has been built and passed as an additional argument. Here, the target language is the explicitly-typed System F, which has a type erasing semantics, hence the type argument `Int` may be dropped while the dictionary argument `OrdInt` is retained: the code that is actually executed is thus `search OrdInt 1 [1; 2; 3]` (where type information has been stripped off `OrdInt` itself).

### 7.2.2 The definition of Mini Haskell

Class declarations and instance definitions are restricted to the toplevel. Their scope is the whole program. In practice, a program `p` is a sequence of class declarations and instance and function definitions given in any order and ending with an expression. For simplification,
we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

Instance definitions are interpreted recursively and their definition order does not matter. We may assume, w.l.o.g., that instance definitions come after all class declarations. The order of class declaration matters, since they may only refer to other class constructors that have been previously defined.

For sake of simplification, we restrict to single parameter classes. The syntax of MH programs is defined in Figure 7.1. Letter p ranges over source programs. A program p is a sequence \( H_1 \ldots H_p, h_1 \ldots h_q \) \( M \), of class declaration \( H_1 \ldots H_p \), followed by a sequence of instance definitions \( h_1 \ldots h_q \), and ending with an expression \( M \).

A class declaration \( H \) is of the form \texttt{class } \( \vec{P} \Rightarrow K \alpha \{ \rho \} \). It defines a new class (constructor) \( K \), parametrized by \( \alpha \). Every class (constructor) \( K \) must be defined by one and only one class declaration. So we may say that \( H \) is the declaration of \( K \) and write \( H \overset{\cdot}{\rightarrow} K \).

Letter \( u \) ranges over overloaded symbols, also called methods. The row \( \rho \) of the form \( u_1 : \tau_1, \ldots u_n : \tau_n \) declares overloaded symbols \( u_i \) of class \( K \). An overloaded symbol cannot be declared twice in a program; it cannot be repeated twice in the same class (hence the map \( i \mapsto u_i \) is injective) and cannot be declared in two different classes. The row \( \rho \) (and thus each of its field type \( \tau_i \)) must not contain any other free variable than \( \alpha \).

The class depends on a sequence of subclasses \( \vec{P} \) of the form \( K_1 \alpha, \ldots K_n \alpha \), which is called a typing context. Each clause \( K_i \alpha \) can be read as an assumption “given an instance of class \( K_i \) at type \( \alpha \)” and \( \vec{P} \) as the conjunction of these assumptions. We say that classes \( K_i \)'s are superclasses of \( K \) which we write \( K_i \prec K \). They must have been previously defined. This ensures that the relation \( \prec \) is acyclic. We require that all \( K_i \)'s are independent, i.e. there doa not exists \( i \) and \( j \) such that \( K_j \prec K_i \).

An instance definition \( h \) is of the form \texttt{inst } \( \forall \vec{\beta}. \vec{P} \Rightarrow K (G \vec{\beta}) \{ r \} \). It defines an instance of a class \( K \) at type \( G \vec{\beta} \) where \( G \) is a datatype constructor, i.e. neither an arrow type nor a class constructor. A class constructor \( K \) may appear in \( Q \) but not in \( \tau \). An instance definition \texttt{defines} the methods of a class at the required type: \( r \) is a record of methods.
An instance definition is also parametrized by a typing context \( \vec{\beta} \) of the form \( K_1 \alpha_1, \ldots, K_k \alpha_k \) where variables \( \alpha_i \)'s are included in \( \vec{\beta} \). This typing context is is not related to the typing context of its class declaration \( H_K \), but to the set of classes that the implementations of the methods depend on.

**Restrictions** The restriction to types of the form \( K' \alpha' \) in typing contexts and class declarations, and to types of the form \( K'(G' \vec{\alpha'}) \) in instances are for simplicity. Generalization are possible and discussed later (§7.4).

### 7.2.3 Semantics of Mini Haskell

The semantics of Mini Haskell is given by elaborating source programs into System F extended with record types and recursive definitions. Record types are provided as data types. They are used to represent dictionaries. Record labels are used to encode overloaded identifiers \( u \). We may use overloaded symbols as variables as well: this amounts to reserving a subset of variables \( x_u \) indexed by overloaded symbols and writing \( u \) as a shortcut for \( x_u \). We use letter \( N \) instead of \( M \) for elaborated terms, to distinguish them from source terms. For convenience, we write \( \Rightarrow \) in System F as an alias for \( \to \), which we use when the argument is a (record representing a) dictionary. Type schemes in the target language take the form described on Figure 7.1. Notice that types \( T \) are stratified: they are either dictionary types \( K \tau \) or a regular type \( \tau \) that does not contain dictionary types.

**Class declaration** The elaboration of a class declaration \( H_K \) of the form class \( K_1 \alpha, \ldots, K_n \alpha \Rightarrow K \alpha \{ \rho \} \) consists of several parts. It first declares a record type that will be used as a dictionary to carry both the methods and the dictionaries of its immediate superclasses. A class need not contain subdictionaries recursively, since if \( K_j \prec K_i \), then a dictionary for \( K_i \) already contains a sub-dictionary for \( K_j \), to which \( K_i \) has access via \( K_i \) so it does need not have one itself. The row \( \rho \) of the class definition only lists the class methods. Hence, we extend it with fields for sub-dictionaries and define the record type:

\[
K \alpha \approx \{ \rho^K \}
\]

where \( \rho^K \) is \( u^K_{K_1} : K_1 \alpha, \ldots, u^K_{K_n} : K_n \alpha, \rho \).

This record type declaration is collected to appear in the program *prelude*.

Then, for each \( u : T_u \) in \( \rho^K \), we define the program context:

\[
\mathcal{R}_u \triangleq \text{let } u : \sigma_u = N_u \text{ in } []
\]

where \( \sigma_u \triangleq \forall \alpha. K \alpha \Rightarrow T_u \) and \( N_u \triangleq \Lambda \alpha. \lambda z : K \alpha. (z.u) \)

Let the composition \( \mathcal{R}_1 \circ \mathcal{R}_2 \) of two contexts be the context \( \mathcal{R}_1[\mathcal{R}_2] \) obtained by placing \( \mathcal{R}_2 \) in the hole of \( \mathcal{R}_1 \). The elaboration \([H_K]\) of a single class declaration \( H_K \) is the composition:

\[
[H_K] \triangleq \mathcal{R}_{u_1} \circ \ldots \mathcal{R}_{u_n}
\]

where \( K \alpha \approx \{ u_1 : T_1, \ldots, u_n : T_n \} \).
that defines accessors for each field of the class dictionary. We also define the typing environment \( \Gamma_H \) as an abbreviation for \( u_1: \sigma_{u_1}, \ldots, u_n: \sigma_{u_n} \).

The elaboration \([H_1 \ldots H_p]\) of all class definitions is the composition \([H_1] \circ \ldots \circ [H_p]\) of the elaboration of each. We also define \( \Gamma_{H_1 \ldots H_n} \) as the concatenation \( \Gamma_{H_1} \ldots \Gamma_{H_n} \) of individual typing environments.

**Instance definition** In an instance declaration \( h \) of the form \texttt{inst } \forall \vec{\beta}. \ P \Rightarrow K (G \vec{\beta}) \{ r \},

The typing context \( \vec{P} \) describes the dictionaries that must be available on type parameters \( \vec{\beta} \) for constructing the dictionary \( K (G \vec{\beta}) \), but that cannot yet be built because they depend on some unknown type \( \beta \) in \( \vec{\beta} \).

As mentioned above \( \vec{P} \) is not related to the typing context of the class declaration \( H_K \). To see this, assume that class \( K' \) is an immediate superclass of \( K \), so that the creation of the dictionary \( K \alpha \) requires the existence of a dictionary \( K' \alpha \); then, an instance declaration \( K \ G \) (where \( G \) is nullary) need not be parametrized over a dictionary of type \( K' \ G \), as either such a dictionary can already be built, hence the instance definition does not require it, or it will never be possible to build one, as instance definitions are recursively defined so all of them are already visible—and the program must be rejected.

We restrict typing context \( K_1 \alpha_1, \ldots, K_k \alpha_k \) to canonical ones defined as satisfying the two following conditions: (1) \( \alpha_i \) is some \( \beta_j \) in \( \vec{\beta} \); and (2) if \( K_i < K_j \) or \( K_j < K_i \) or \( K_i = K_j \). then \( \alpha_i \) and \( \alpha_j \) are different. The latter condition avoids having two dictionaries \( K_i \beta \) and \( K_j \beta \) when, e.g., \( K_i < K_j \) since the former is contained in the latter.

The elaboration of an instance declaration \( h \) is a triple \((z_h, N^h, \sigma_h)\) where \( z_h \) is an identifier to refer to the elaborated body \( N^h \) of type

\[
\sigma_h \triangleq \forall \vec{\beta}_1 \ldots \vec{\beta}_p. K_1 \alpha_1 = \ldots K_k \alpha_k \Rightarrow K (G \vec{\beta})
\]

(Variables \( \alpha_1, \ldots, \alpha_k \) are among \( \beta_1, \ldots, \beta_p \) and may contain repetitions, as explained above.)

The expression \( N^h \) builds a dictionary of type \( K (G \vec{\beta}) \), given \( p \) dictionaries (where \( p \) may be zero) of respective types \( K_1 \beta_1, \ldots, K_k \beta_k \) and is defined as:

\[
N^h \triangleq \Lambda \beta_1. \ldots \Lambda \beta_p. \lambda (z_1:K_1 \alpha_1). \ldots \lambda (z_k:K_k \alpha_k).
\]

\[
\{ u^K_{K_1} = q_1, \ldots, u^K_{K_k} = q_n, u_1 = N^h_1, \ldots, u_m = N^h_m \}
\]

The types of fields are as prescribed by the class definition \( K \), but specialized at type \( G \vec{\beta} \). That is, \( q_i \) is a dictionary expression of type \( K'_i (G \vec{\beta}) \) whose exact definition is postponed until the elaboration of dictionaries in \( \S 7.2.6 \). The term \( N^h_i \) is the elaboration of \( M_i \) where \( u_1 = M_1, \ldots, u_m = M_m \) is \( r \); it is described in the next section (\( \S 7.2.4 \)). For clarity, we write \( z \) instead of \( x \) when a variable binds a dictionary or a function building a dictionary. Notice that the expressions \( q_i \) and \( N^h_i \) sees the type variables \( \beta_1, \ldots, \beta_p \) and the dictionary parameters \( z_1 : K_1 \alpha_1, \ldots, z_k : K_k \alpha_k \).
The elaboration of all instance definitions is the program context:

\[ \llbracket \tilde{h} \rrbracket \triangleq \text{let rec } (\tilde{z}_h : \tilde{\sigma}_h) = \tilde{N}_h \text{ in } [] \]

that recursively binds all instance definitions in the hole.

**Program** Finally, the elaboration of a complete program \( H \tilde{h} M \) is

\[ \llbracket H \tilde{h} M \rrbracket \triangleq (\llbracket H \rrbracket \circ \llbracket \tilde{h} \rrbracket \llbracket M \rrbracket) = \text{let } \tilde{u} : \tilde{\sigma}_u = \tilde{N}_u \text{ in let rec } (\tilde{z}_h : \tilde{\sigma}_h) = \tilde{N}_h \text{ in } N \]

Hence, the expression \( N \), which is the elaboration of \( M \), and all expressions \( N_h \) are typed (and elaborated) in the environment \( \Gamma \tilde{H} \tilde{h} \) equal to \( \Gamma_{\tilde{H}} \), \( \Gamma_{\tilde{h}} \): the environment \( \Gamma_{\tilde{H}} \) declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols) while the environment \( \Gamma_{\tilde{h}} \), declares functions to build dictionaries.

### 7.2.4 Elaboration of expressions

The elaboration of expressions is defined by a judgment \( \Gamma \vdash M \leadsto N : \sigma \) where \( \Gamma \) is a System F typing context, \( M \) is the source expression, \( N \) is the elaborated expression and \( \sigma \) its type in \( \Gamma \). In particular, \( \Gamma \vdash M \leadsto N : \sigma \) implies \( \Gamma \vdash N : \sigma \) in System F.

We write \( q \) for dictionary terms, which are the following subset of System-F terms:

\[ q ::= u \mid z \mid q \tau \mid q q \]

Variables \( u \) and \( z \) are just particular cases of variables \( x \). Variable \( u \) is used for methods (and access to subdictionaries), while variable \( z \) is used for dictionary parameters and for class instances, *i.e.* dictionaries or functions building dictionaries.

The rules for elaboration of expressions are described in Figure 7.2. Most of them just wrap the elaboration of their sub-expressions. In rule `LET`, we require \( \sigma \) to be canonical, *i.e.* of the form \( \forall \tilde{\alpha}. \tilde{P} \Rightarrow T \) where \( \tilde{P} \) is itself empty or canonical (see page 153). Rules `APP` and `ABS` do not apply to overloaded expressions of type \( \sigma \) but only to simple expressions of type \( \tau \).

The interesting rules are the elaboration of overloaded expressions, and in particular of missing abstractions (Rule `OABS`) and applications (Rule `OAPP`) of dictionaries. Rule `OABS` pushes dictionary abstractions in the context \( \Gamma \) as prescribed by the expected type. On the opposite, Rule `OAPP` searches for an appropriate dictionary-building function and applies it to the required sub-directionary.

The premise \( \Gamma \vdash q : Q \) of rule `OAPP` also triggers the elaboration of dictionaries. This judgment is just the typability in System F—but restricted to dictionary expressions. That is, it searches for a well-typed dictionary expression. The restriction to dictionary expressions ensures that under reasonable conditions the search is decidable—and coherent. The elaboration of dictionaries reads the typing rules of System F restricted to dictionaries as an algorithm, where \( \Gamma \) and \( Q \) are given and \( q \) is inferred. This is described in detail in §7.2.6.
7.2. **MINI HASKELL**

<table>
<thead>
<tr>
<th><strong>VAR</strong></th>
<th><strong>INST</strong></th>
<th><strong>GEN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x : \sigma \in \Gamma )</td>
<td>( \Gamma \vdash M \rightsquigarrow N : \forall \alpha . \sigma )</td>
<td>( \Gamma , \alpha \vdash M \rightsquigarrow N : \sigma )</td>
</tr>
<tr>
<td>( \Gamma \vdash x \rightsquigarrow x : \sigma )</td>
<td>( \Gamma \vdash M \tau \rightsquigarrow N \tau : [\alpha \mapsto \tau ]\sigma )</td>
<td>( \Gamma \vdash \Lambda \alpha . M \rightsquigarrow \Lambda \alpha . N : \forall \alpha . \sigma )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>LET</strong></th>
<th><strong>ABS</strong></th>
<th><strong>OABS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash M_1 \rightsquigarrow N_1 : \sigma )</td>
<td>( \Gamma \vdash x : \sigma \vdash M_2 \rightsquigarrow N_2 : \tau )</td>
<td>( \Gamma \vdash x : \tau' \vdash M \rightsquigarrow N : \tau )</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{let } x : \sigma = M_1 \text{ in } M_2 \rightsquigarrow \text{let } x : \sigma = N_1 \text{ in } N_2 : \tau )</td>
<td>( \Gamma \vdash \lambda x : \tau' . M \rightsquigarrow \lambda x : \tau' . N : \tau' \rightsquigarrow \tau )</td>
<td>( \Gamma \vdash x : Q \vdash M \rightsquigarrow N : \sigma \quad x \neq M )</td>
</tr>
<tr>
<td>( \Gamma \vdash M_1 \rightsquigarrow N_1 : \tau_1 \rightsquigarrow \tau_2 )</td>
<td>( \Gamma \vdash M_2 \rightsquigarrow N_2 : \tau_2 )</td>
<td>( \Gamma \vdash \lambda x : Q . M \rightsquigarrow \lambda x : Q . N \rightsquigarrow \sigma )</td>
</tr>
<tr>
<td>( \Gamma \vdash M_1 M_2 \rightsquigarrow N_1 N_2 : \tau_1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>OAPP</strong></th>
<th><strong>OAPP</strong></th>
<th><strong>OAPP</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma )</td>
<td>( \Gamma \vdash q : Q )</td>
<td>( \Gamma \vdash M \rightsquigarrow N q : \sigma )</td>
</tr>
</tbody>
</table>

Figure 7.2: Elaboration of expressions

By construction, elaboration produces well-typed expressions: that is \( \Gamma \vdash M \rightsquigarrow N : \tau \) implies that is \( \Gamma \vdash N : \tau \).

### 7.2.5 Summary of the elaboration

An instance declaration \( h \) of the form:

\[
\text{inst } \forall \vec{\beta}, K_1 \alpha_1, \ldots K_k \alpha_k \Rightarrow K \bar{\tau} \{ u_1 = M_1, \ldots ; u_m = M_m \}
\]

is translated into

\[
\lambda (z_1 : K_1 \alpha_1) \ldots \lambda (z_p : K_k \alpha_k) \{ u^K_{K_1} = q_1, \ldots u^K_{K_k} = q_n, \ u_1 = N_1, \ldots u_m = N_m \}
\]

where \( u^K_{K_i} : \tau_i \) are the superclasses fields, \( \Gamma^h \) is \( \vec{\beta}, K_1 \alpha_1, \ldots K_k \alpha_k \), and the following elaboration judgments \( \Gamma^h \vdash q_i : \tau_i \) and \( \Gamma^h \vdash M_i \rightsquigarrow N_i : \tau_i \) hold. Finally, given the program \( p \) equal to \( \tilde{h} \tilde{h} M \), we elaborate \( M \) as \( N \) such that \( \Gamma^h \vdash M \rightsquigarrow N : \forall \bar{\alpha} . \tau \).

Notice that \( \forall \bar{\alpha} . \tau \) is an unconstrained type scheme. Otherwise, \( N \) could elaborate into an abstraction over dictionaries, which could turn a computation into a function that is not reduced: this would not preserve the intended semantics.

More generally, we must be careful to preserve the intended semantics of source programs. For this reason, in a call-by-value setting, we must not elaborate applications into abstractions, since this could delay and perhaps duplicate the order of evaluations. We just pick the obvious solution, that is to restrict rule \( \text{LET} \) so that either \( \sigma \) is of the form \( \forall \bar{\alpha} . \tau \) or \( M_1 \) is a value or a variable.
In a language with a call-by-name semantics, an application is not evaluated until it is needed. Hence adding an abstraction in front of an application should not change the evaluation order $M_1 M_2$. We must in fact compare:

$$\begin{align*}
\text{let } x_1 = \lambda y. \text{let } x_2 = V_1 V_2 \text{ in } M_2 \text{ in } [x_1 \mapsto x_1 q]M_1 \\
\text{let } x_1 = \text{let } x_2 = \lambda y. V_1 V_2 \text{ in } [x_2 \mapsto x_2 q]M_2 \text{ in } M_1
\end{align*}$$

(1) (2)

The order of evaluation of $V_1 V_2$ is preserved. However, the Haskell language is call-by-need and not call-by-name! Hence, applications are delayed as in call-by-name but shared and only reduced once. The application $V_1 V_2$ will be reduced once in (1), but as many times as there are occurrences of $x_2$ in $M_2$ in (2).

The final result will still be the same in both cases if the language has no side effects, but the intended semantics may be changed regarding the complexity.

**Coherence** The elaboration may fail for several reasons: The input expression may not obey one of the restrictions we have requested; a typing may occur during elaboration of an expression; or or some dictionary cannot be build. If elaboration fails, the program $p$ is rejected, of course.

When the elaboration of $p$ succeeds, it should return a term $\llbracket p \rrbracket$ that is well-typed in $F$ and that defines the semantics of $p$. However, although terms are explicitly-typed, their elaboration may not be unique! Indeed, they might be several ways to build dictionaries of some given type, as we shall see below (§7.2.6).

We may distinguish two situations: in the worst case, a source program may elaborate to several completely unrelated programs; in the better case, all possible elaborations may in fact be equivalent programs: we say that the elaboration is coherent and the programs has a deterministic semantics given by any of its elaboration.

Opening a parenthesis, what does it mean for programs be equivalent? There are several notions of program equivalence:

- If programs have a denotational semantics, the equivalence of programs should be the equality of their denotations.

- As a subcase, two programs having a common reduct should definitely be equivalent. However, this will in general not be complete: values may contain functions that are not identical, but perhaps reduce to the same value whenever applied to equivalent arguments.

- This leads to the notion of observational equivalence. Two expressions are observationally equivalent (at some observable type, such as integers) if their are indistinguishable whenever they are put in arbitrary (well-typed) contexts of the observable type.

End of parenthesis.
For instance, two different elaboration algorithms that consistently change the representation of dictionaries (e.g. by ordering records in reverse order), may be equivalent if we cannot observe the representation of dictionaries.

Returning to the coherence problem, the only source of non-determinism in Mini Haskell is the elaboration of dictionaries. Hence, to ensure coherence, it suffices that two dictionary values of the same type are always equal. This does not mean that there is a unique way of building dictionaries, but that all ways are equivalent as they eventually return the same dictionary.

### 7.2.6 Elaboration of dictionaries

The elaboration of dictionaries is based on typing rules of System F—but restricted to a subset of the language. The relevant typing rules are given in Figure 7.3. However, elaboration significantly differs from type inference since the judgment $\Gamma \vdash q : Q$ is used for inferring $q$ rather than $\tau$. The judgment can be read as: in type environment $\Gamma$, a dictionary of type $Q$ can be constructed by the dictionary expression $q$. As for type inference, elaboration of dictionaries is simplified by finding an appropriate syntax-directed presentation of the typing rules—but directed by the structure of the type of the expected dictionary instead of expressions.

Elaboration is also driven by the bindings available in the typing environment. These may be dictionary constructors $z^h$, given by instance definitions; dictionary accessors $u^K$, given by class declarations; dictionary arguments $z$, given by the local typing context. This suggests the presentation of the typing rules in Figure 7.4.

**Dictionary values** Let us first consider the elaboration of dictionary values, i.e. dictionary expressions that do not use dictionary parameters or projections. Thus, their derivation may only use $[$D-OVAR-INST$]$. They are typed in the environment $\Gamma^h$, which does not contain free type variables. They actually do not access dictionaries, and only use the environment $\Gamma$. Hence, all occurrences of $[$D-OVAR-INST$]$ are of the form:

$$
\begin{align*}
\text{D-OVAR-INST} & \quad z : \forall \beta. P_1 \Rightarrow \ldots P_n \Rightarrow K (\tilde{G} \tilde{\beta}) \in \Gamma^h & \Gamma^h \vdash q_i : [\tilde{\beta} \mapsto \tilde{\tau}]P_i \\
\end{align*}
$$

Hence, all occurrences of $[$D-OVAR-INST$]$ are of the form:

$$
\begin{align*}
\text{D-OVAR-INST} & \quad z : \forall \beta. P_1 \Rightarrow \ldots P_n \Rightarrow K (\tilde{G} \tilde{\beta}) \in \Gamma^h & \Gamma^h \vdash q_i : [\tilde{\beta} \mapsto \tilde{\tau}]P_i \\
\end{align*}
$$

\[ \Gamma^h \vdash z \tilde{\tau} \tilde{q} : K (G \tilde{\tau}) \]
and the premise $\Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}] P_i$ is itself recursively built in the same way with this single rule. This rule can be read as a recursive definition, where $\Gamma$ is constant, $Q$ is the input type of the dictionary, and $q$ is the output dictionary. This reading is deterministic if there is no choice in finding $z : \forall \vec{\beta}. P_1 \Rightarrow \ldots P_n \Rightarrow K (G \vec{\beta})$ in $\Gamma$. The binding $z$ can only be a binding $z^h$ introduced as the elaboration of some class instance $h$ at type $\Gamma \vec{\beta}$. Hence, it suffices that instance definitions never overlap for $z^h$ to be uniquely determined; if recursively each $q_i$ is unique, then $z \vec{\tau} \vec{q}$ also is. Under this hypothesis, the elaboration is always unique and therefore coherent.

**Definition 4 (Overlapping instances)** Two instances inst $\forall \vec{\beta}_1. P \Rightarrow K (G_1 \vec{\beta}_1) \{r_1\}$ and inst $\forall \vec{\beta}_2. P \Rightarrow K (G_2 \vec{\beta}_2) \{r_2\}$ of a class $K$ overlap if the type schemes $\forall \vec{\beta}_1. K (G_1 \vec{\tau}_1)$ and $\forall \vec{\beta}_2. K (G_2 \vec{\tau}_2)$ have a common instance, i.e. in the current setting, if $G_1$ and $G_2$ are equal.

Overlapping instances are an inherent source of incoherence, as it means that for some type $Q$ (in the common instance), a dictionary of type $Q$ may (possibly) be built using two different implementations.

**Dictionary expressions** Dictionary expressions may compute on dictionaries: they may extract sub-dictionaries or build new dictionaries from other dictionaries received as argument. Indeed, in overloaded code, the exact type is not fully known at compile type, hence dictionaries must be passed as arguments, from which superclass dictionaries may be extracted (actually must be extracted, as we forbade to pass a class and one of its super class dictionaries simultaneously).

Dictionaries are typically typed in the typing environment $\Gamma^H_h, \Gamma^h$ where $\Gamma^h$ binds the local typing context, i.e. assumptions $z : K' \vec{\beta}$ about dictionaries received as arguments. Hence, rules $D\text{-}\text{Proj}$ and $D\text{-}\text{Var}$ may now apply, i.e. the elaboration of expressions uses the three rules of 7.4. This can still be read as a backtracking proof search algorithm. The proof search always terminates, since premises always have strictly smaller $Q$ than the conclusion when using the lexicographic ordering of the height of $\tau$ and then the reverse order of class inheritance: when no rule applies, the search fails; when rule $D\text{-}\text{Var}$ applies, the search ends
with a successful derivation; when rule \textbf{D-PROJ} applies, the premise is called with a smaller problem since the height is unchanged and \( K' \prec K \); when \textbf{D-OVAR-INST} applies, the premises are called at type \( K_i \tau_j \) where \( \tau_j \) is subtype of \( \bar{\tau} \), hence of a strictly smaller height.

**Non determinism** However, non-overlapping of class instances is no more sufficient to prevent from non determinism. For instance, the introductory example of §7.2.1 defines two instances \( \text{EqInt} \) and \( \text{OrdInt} \) where the later contains an instance of the former. Hence, a dictionary of type \( \text{EqInt} \) may be obtained, either directly as \( \text{EqInt} \), or indirectly as \( \text{Eq OrdInt} \), by projecting the \( \text{Eq} \) sub-dictionary of class \( \text{Ord Int} \). In fact, the latter choice could then be reduced at compile time and be equivalent to the first one.

One could force more determinism by fixing a strategy for elaboration. Restrict the use of rule \textbf{D-PROJ} to cases where \( Q \) is \( P \)—when \textbf{D-OVAR-INST} does not apply. However, since the two elaborations paths are equivalent, the extra flexibility is harmless and may perhaps be useful freedom for the compiler.

**Example of elaboration** In our introductory example, the typing environment \( \Gamma_{HH} \) is (we remind both the informal and formal names of variables):

\[
\begin{align*}
equal & \triangleq u_{equal} : \forall \alpha. \text{Eq} \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{bool}, \\
\text{EqInt} & \triangleq z_{\text{int}}^{\text{Eq}} : \text{Eq int} \\
\text{EqList} & \triangleq z_{\text{List}}^{\text{Eq}} : \forall \alpha. \text{Eq} \alpha \Rightarrow \text{Eq} (\text{list} \alpha) \\
\text{EqOrd} & \triangleq u_{\text{Ord}}^{\text{Eq}} : \forall \alpha. \text{Ord} \alpha \Rightarrow \text{Eq} \alpha \\
\text{lt} & \triangleq u_{\text{lt}} : \forall \alpha. \text{Ord} \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{bool}
\end{align*}
\]

When elaborating the body of the \textit{search} function, we have to infer a dictionary for \( \text{EqOrd} \ X \text{OrdX} \) in the local context \( X, \text{OrdX} : \text{Ord} X \). Using formal notations, dictionaries are typed in the environment \( \Gamma \) equal to \( \Gamma_0, \alpha, z : \text{Ord} \alpha \) and \( \text{EqOrd} \) is \( u_{\text{Eq}}^{\text{Ord}} \). We have the following derivation:

\[
\begin{array}{c}
\text{D-OVAR-INST} \\
\Gamma \vdash z : u_{\text{Ord}}^{\text{Eq}} : \text{Ord} \alpha \rightarrow \text{Eq} \alpha \\
\text{D-PROJ} \\
\Gamma \vdash z : \text{Ord} \alpha
\end{array}
\Rightarrow
\begin{array}{c}
\text{D-VAR} \\
\Gamma \vdash z : \text{Ord} \alpha \\
\text{D-PROJ} \\
\Gamma \vdash u_{\text{Eq}}^{\text{Ord}} \alpha z : \text{Eq} \alpha
\end{array}
\]

### 7.3 Implicitly-typed terms

Our presentation of Mini Haskell is explicitly typed. Since we remain within an ML-like type system where type schemes are not first-class, we may leave some type information implicit. But how much? Class declarations define both the structure of dictionaries—a record type definition and its accessors—and the type scheme of overloaded symbols. Since, we inferring type schemes is out of the scope of ML-like type inference, class declarations
must remain explicit. Instance definitions are turned into recursive polymorphic definitions, 
which in ML require type scheme annotations. So they instance definitions also remain 
explicit. Fortunately, all remaining core language expressions, i.e. the body of instance 
definitions and the final program expression can be left implicit.

For instance, the example program in the introduction can be rewritten more concisely.

```haskell
class Eq (X) { equal : X → X → Bool }
inst Eq (Int) { equal = (==) }
inst Eq (Char) { equal = (==) }
inst Λ(X) Eq (X) ⇒ Eq (List (X))
  { Eq = λ(l₁) λ(l₂) match l₁, l₂ with
    | [],[] → true | [],_ | _,[] → false
    | h₁::t₁, h₂::t₂ → Eq h₁ h₂ && Eq t₁ t₂ }

class Eq (X) ⇒ Ord (X) { lt : X → X → Bool }
inst Ord (Int) { lt = (<) }

let rec search x l = match l with [] → false | h::t → equal x h || search x t
let b = search Int 1 [1; 2; 3];;
```

The missing type information can be built by type inference.

Type inference To perform type inference in Mini Haskell, the idea is to see dictionary 
types K τ, which can only appear in type schemes and not in types, as a type constraint 
to mean “there exists a dictionary of type K α”. That is, we may read the type scheme 
∀α. P ⇒ τ as the constraint type scheme ∀α[P].τ where P is seen as a type predicate, say 
a dictionary predicate. Therefore, we extend constraints with dictionary predicates:

\[ C ::= \ldots | K τ \]

On ground types, a constraint K t is satisfied if one can build a dictionary of type K t in 
the initial environment ΓHH (that contains all class and instance declarations)—formally, 
if there exists a dictionary expression q such that ΓHH ⊢ q : K t. Then satisfiability of 
class-membership constraints is (with its unfolded version on the right):

\[
\begin{align*}
\text{INSTANCE} & \\
K φτ & \quad \text{INSTANCE} & \\
φ ⊢ K τ & \quad ΓHH ⊢ ρ : K φτ & \quad φ ⊢ K τ
\end{align*}
\]

We use entailment to reason with class-membership constraints. For every class declaration 
class K₁ α₁, ... Kₙ αₙ ⇒ K α {ρ}, we have:

\[ K α ⊢ K₁ α₁ ∧ ... Kₙ αₙ \]  

(K1)

This rule allows to decompose any set of simple constraints into a canonical one.

Proof: Assume φ ⊢ K α, i.e. by Rule INSTANCE ΓHH ⊢ q : K (φα) for some dictionary q. 
From the class declaration in ΓHH, we know that K α is a record type definition that contains
fields \( u^K_i \) of type \( K_i \alpha_i \). Hence, the dictionary value \( q \) contains field values of types \( K_i (\phi \alpha) \). Therefore, we have \( \phi \vdash K_i \alpha \) for all \( i \) in \( 1..n \), which implies \( \phi \vdash K_1 \alpha \land \ldots \land K_n \alpha \).

For every instance definition \( \text{inst} \ \forall \vec{\beta}. K_i \beta_1, \ldots, K_p \beta_p \Rightarrow K (G \vec{\beta}) \{r\} \), we have

\[
K (G \vec{\beta}) \equiv K_1 \beta_1 \land \ldots \land K_p \beta_p
\]  

(K2)

This rule allows to decompose any class constraint into a conjunction of simple constraints (i.e. of the form \( K \alpha \)).

Proof: Let \( h \) be the above instance definition. We proof both directions separately:

Case \( \vdash \): Assume \( \phi \vdash K_i \beta_i \) for \( i \) in \( \{1, \ldots, p\} \). By Rule [INSTANCE] for each \( i \), there exists a dictionary \( q_i \) such that \( \Gamma_{\vec{H}} \vdash q_i : K_i (\phi \beta_i) \). Hence, \( \Gamma_{\vec{H}} \vdash x_{\vec{h}} \beta q_1 \ldots q_p : K (G (\phi \vec{\beta})) \), i.e. by Rule [INSTANCE] \( \phi \vdash K (G \vec{\beta}) \).

Case \( \models \): Assume, \( \phi \vdash K (G \vec{\beta}) \). i.e. there exists a dictionary \( q \) such that \( \Gamma_{\vec{H}} \vdash q : K (G (\phi \vec{\beta})) \). By inversion of typing (and non-overlapping of instance declarations), the only way to build such a dictionary is by an application of \( z_{\vec{h}} \). Hence, \( q \) must be of the form \( x_{\vec{h}} \beta q_1 \ldots q_p \) with \( \Gamma_{\vec{H}} \vdash q_i : K_i (\phi \beta_i) \). By Rule [INSTANCE] this means \( \phi \vdash K_i \beta_i \) for every \( i \), which implies \( \phi \vdash K_1 \beta_1 \land \ldots \land K_p \beta_p \).

Notice that the equivalence (K2) still holds in an open-world assumption where new instance clauses may be added later, because another future instance definition cannot overlap with existing ones.

If class instances may overlap, the \( \models \) direction does not hold anymore; the rewriting rule:

\[
K (G \vec{\beta}) \rightarrow K_1 \beta_1 \land \ldots \land K_p \beta_p
\]

remains sound (the inverse entailment holds, and thus type inference still infer sound typings), but it is incomplete (type inference could miss some typings).

We also use the following equivalence: for every class \( K \) and type constructor \( G \) for which there is no instance of \( K \):

\[
K (G \vec{\beta}) \equiv \text{false}
\]  

(K3)

This rule allows to report failure as soon as a constraint of the form \( K (G \vec{\tau}) \) for which there is not instance of \( K \) for \( G \) appears.

Proof: The \( \vdash \) direction is a tautology, so it suffices to prove the \( \models \) direction. By contradiction. Assume \( \phi \vdash K (G \vec{\beta}) \). This implies the existence of a dictionary \( q \) such that \( \Gamma_{\vec{H}} \vdash q : K (G (\phi \vec{\beta})) \). Then, there must be some \( x_{\vec{h}} \) in \( \Gamma \) whose type scheme is of the form \( \forall \vec{\beta}. P \Rightarrow K (G \vec{\beta}) \), i.e. there must be an instance of class \( K \) for \( G \).
CHAPTER 7. OVERLOADING

Notice that the equivalence is only an inverse entailment in an open world assumption: when there is not instance of $K$ at type $G$, the rewriting rule $K(G \beta) \rightarrow false$ remains sound, but it is incomplete.

We are now fully equipped for type inference. Constraint generation is unchanged: see Figure 5.6. A constraint type scheme can then always be decomposed into one of the form $\forall \tilde{\alpha}[P_1 \land P_2]. \tau$ where $\text{ftv}(P_1) \in \tilde{\alpha}$ and $\text{ftv}(P_2) \neq \tilde{\alpha}$. The constraints $P_2$ can then be extruded to the enclosing context if any, so that we are just left with $P_1$, and thus a well-formed type scheme $\forall \tilde{\alpha}. \tilde{P} \Rightarrow \tau$ with a typing context $\tilde{P}$.

To check well-typedness of a program $\tilde{H} \tilde{h} \tilde{a}$, we must check that: each expression $a^h$ and the expression $a$ are well-typed, in the environment used to elaborate them. This amounts to checking:

- $\Gamma_{\tilde{H}h}, \Gamma^h \vdash a^h : \tau^h$ where $\tau^h$ is given. That is, that $\text{def} \Gamma_{\tilde{H}h}, \Gamma^h \in \{a^h\} \leq \tau^h \equiv \text{true}$ holds;
- $\Gamma_{\tilde{H}h} \vdash a : \tau$ for some $\tau$. That is, that $\text{def} \Gamma_{\tilde{H}h} \in \exists \alpha. \{a\} \leq \alpha \equiv \text{true}$ holds.

However, typechecking is not sufficient: type reconstruction should also return an explicitly-typed term $M$ than can in turn be elaborated into some term $N$ of System $F$, i.e. such that $\Gamma \vdash a \leadsto M : \tau$.

**Type reconstruction** Type reconstruction can be performed as described in §5.3.4 by keeping persistent constraints during resolution. As in ML, there may be several ways to reconstruct programs, which we may solve by requesting explicitly-typed terms to be canonical and principal.

**Coherence** When the source language is implicitly-typed, the elaboration from the source language into System $F$ code is the composition of type reconstruction with elaboration of explicitly typed terms.

Hence, even though the elaboration is coherent for explicitly-typed terms, this may not be true for implicitly-typed terms. There are two potential problems:

- The language has principal constrained type schemes, but the elaboration requests unconstrained type schemes.
- Ambiguities could be hidden (and missed) by non principal type reconstructions.

**Toplevel unresolved constraints** The restrictions we put on class declarations and instance definitions ensure that the type system has principal constrained schemes (and principal typing reconstructions).

However, this does not imply that there are principal *unconstrained* type schemes. For example, assume that the principal constrained type scheme is $\forall \alpha[K \alpha]. \alpha \rightarrow \alpha$ and the typing environment contains two instances of $K G_1$ and $K G_2$ of class $K$. Constraint-free
instances of this type scheme are $G_1 \rightarrow G_1$ and $G_2 \rightarrow G_2$ but $\forall \alpha. \alpha \rightarrow \alpha$ is certainly not one. Not only neither choice is principal, but worse, the two choices would elaborate in expressions with different (and non-equivalent) semantics. Elaboration should fail in such cases.

This problem may appear while typechecking the final expression $a$ in $\Gamma_{\tilde{H}h}$ that request an unconstrained type scheme $\forall \alpha. \tau$ It may also occur when typechecking the body of an instance definition $h$, which requests an explicit type scheme $\forall \tilde{\beta}[\tilde{Q}]. \tau$ in $\Gamma_{\tilde{H}h}$ or, equivalently, a type $\tau$ in $\Gamma_{\tilde{H}h}, \tilde{\beta}, \tilde{Q}$. Consider, for example:

```plaintext
class Num (X) { 0 : X, (+) : X \rightarrow X \rightarrow X }
inst Num Int { 0 = Int.(0), (+) = Int.(+) }
inst Num Float { 0 = Float.(0), (+) = Float.(+) }
let zero = 0 + 0;
```

The type of $zero$ or $zero + zero$ is $\forall \alpha[Num \alpha].\alpha$ while several class instances are possible for $Num X$. The semantics of the program is thus undetermined. Another example is:

```plaintext
class Readable (X) { read : descr \rightarrow X }
inst Readable (Int) { read = read_int }
inst Readable (Char) { read = read_char }
let v = read (open_in())
```

The type of $v$ is $\forall \alpha[Readable \alpha].unit \rightarrow \alpha$—and several classes are possible for $Readable \alpha$. This program is also rejected.

**Inaccessible constraint variables** In the previous examples, the incoherence arise from the obligation to infer unconstrained toplevel type schemes. A similar problem may occur with isolated constraints in a type scheme. For instance, assume that let $x = a_1$ in $a_2$ elaborates to let $x : \forall \alpha[K \alpha],int \rightarrow int = N_1$ in $N_2$. All applications of $x$ in $N_2$ will lead to an unresolved constraint $K \alpha$ for some fresh $\alpha$ since neither the argument nor the context of this application can determine the value of the type parameter $\alpha$. Still, a dictionary of type $K \tau$ must be given before $N_1$ can be executed.

Although $x$ may not be used in $N_2$, in which case, all elaborations of the expression may be coherent, we may still raise an error, since an unusable local definition is certainly useless, hence probably a programmer’s mistake. The error may then be raised immediately, at the definition site, instead of at every use of $x$.

**The open-world view** When there is a single instance $K G$ for a class $K$ that appears in an unresolved or isolated constraint $K \alpha$, the problem formally disappears, as all possible type reconstructions are coherent.

However, we may still not accept this situation, for modularity reasons, as an extension of the program with another non-overlapping correct instance declaration would make the program become ambiguous.
Formally, this amounts to saying that the program must be coherent in its current form, but also in all possible extensions with well-typed class definitions. This is taking an open-world view.

**On the importance of principal type reconstruction** A source of incoherence is when some class constraint remains undetermined. Some (usually arbitrary) less general elaboration could cover the problem—but the source program would remain incoherent. Hence, in order to detect programs with ambiguous semantics, it is essential that type reconstruction is principal. A program can still be specialized but only after it has been proved coherent. This freedom may actually be very useful for optimizations. Consider for example, the program

\[
\text{let } \text{twice } = \lambda(x) \ x + x \ \text{in} \ \text{twice } (\text{twice } 1)
\]

whose principal type reconstruction is:

\[
\text{let } \text{twice } : \forall(X) [ \text{Num} \ X ] X \to X = \Lambda(X) [\text{Num} \ X] \lambda(x) \ x + x \ \text{in}
\text{twice Int } (\text{twice Int } 1)
\]

This program is coherent. It’s natural elaboration is

\[
\text{let } \text{twice } X \text{ Num}X = \lambda(x : X) \ (x + \text{Num}X) \ x \ \text{in}
\text{twice Int NumInt } (\text{twice Int NumInt } 1)
\]

However, it can also be elaborated to

\[
\text{let } \text{twice } = \lambda(x : \text{Int}) \ x + \text{NumInt} \ x \ \text{in} \ \text{twice } (\text{twice } 1)
\]

avoiding the generalization of twice; moreover, the overloaded application \(\text{plus NumInt}\) can now be statically reduced, leading to:

\[
\text{let } \text{twice } = \lambda(x : \text{Int}) \ x \ \text{Int.(+)} \ x \ \text{in} \ \text{twice } (\text{twice } 1)
\]

**Overloading by return types** All previous ambiguous examples are overloaded by their return types: For instance, in \(0 : X\), the value \(0\) has an overloaded type that is not constraint by the argument; in \(\text{read} : \text{descr} \to X\), the return type is under specified, independently of the type of the argument.

To avoid such cases, Odersky et al. has suggested to prevent overloading by return types by requesting that overloaded symbols of a class \(K\ \alpha\) have types of the form \(\alpha \to \tau\). The above examples would then be rejected by this definition.

In fact, disallowing overloading by return types—in addition to our previous restrictions—suffices to ensure that all well-typed programs are coherent. Moreover, untyped programs can then be given a direct semantics (which of course coincides with the semantics obtained by elaboration). Many interesting examples of overloading actually fits in this restricted subset. However, overloading by returns types is also found useful in several cases and Haskell allows it, using default rules to resolve ambiguities. This is still an arguable design choice in the Haskell community.
7.4 Variations

Changing the representation of dictionaries  An overloaded method call $u$ of a class $K$ is elaborated into an application $u \ q$ of $u$ to a dictionary expression $q$ of class $K$. The function $u$ and the representation of the dictionary are both defined in the elaboration of the class $K$ and need not be known at the call site. This leaves some flexibility in the representation of dictionaries. For example, we have used records to represent dictionaries, but tuples would have been sufficient.

Going one step further, dictionaries need not contain the methods themselves but enough information from which the methods may be recovered. For example, dictionaries may be replaced by a derivation tree that proves the existence of the dictionary. This derivation tree may be concisely represented and passed around instead of the dictionary itself and be used and interpreted at the call site to dispatch to the appropriate implementation of the method. Such an approach has been followed by Furuse (2003b).

This change of representation can also elegantly be explained as a type preserving compilation of dictionaries called concretization and described in Pottier and Gauthier (2006). It is somehow similar to defunctionalization and also requires that the target language is equipped with GADT (Guarded Abstract Data Types).

Multi-parameter type classes  To allow multi-parameter type classes, we may extend
the syntax of class definitions as follows:

$$\text{class } \bar{P} \Rightarrow K \bar{a} \{ \rho \}$$

where free variables of $\bar{P}$ must be bound in $\bar{a}$. The current framework can easily be extended to handle multi-parameter type classes. For example, Collections may be represented by a
type $C$ whose elements are of type $E$ and defined as follows:

$$\text{class } \text{Collection } C E \{ \text{find } : C \rightarrow E \rightarrow \text{Option}(E), \text{add } : C \rightarrow E \rightarrow C \}$$
$$\text{inst } \text{Collection } (\text{List } X) X \{ \text{find } = \text{List.find, add } \Rightarrow \lambda(c)\lambda(e) e::c \}$$
$$\text{inst } \text{Collection } (\text{Set } X) X \{ \ldots \}$$

However, the class Collection does not provide the intended intuition that collections are homogeneous. Indeed, we may define:

$$\text{let } \text{add2 } c x y = \text{add } (\text{add } c x) y$$
$$\text{add2 : } \forall(C, E, E') \text{ Collection } C E, \text{ Collection } C E' \Rightarrow C \rightarrow E \rightarrow E' \rightarrow C$$

This is accepted assuming that collections are heterogeneous. Although, this is unlikely the case, no contradiction can be assumed. However, if collections are indeed homogeneous, no instance of heterogeneous collections will ever be provided and the above code is overly general. As a result, uses of collections have unresolved often parameters, which would be resolved, if we had a way to tell the system that collections are homogeneous.

The solution is to add a clause to say that the parameter $C$ determines the parameter $E$:
class Collection \( C E \mid C \to E \) \{ ... \}

Then, because \( C \) determines \( E \), the two instances \( E \) and \( E' \) must be equal in \( C \). Type dependencies also reduce overlapping between class declarations, since fewer instances of a class make sense. Hence they also allow example that would have to be rejected if type dependencies could not be expressed.

**Associated types**  Associated types are an alternative to functional dependencies. They allow a class to declare its own type functions. Correspondingly, instance definitions must provide a definition for all associated types—in addition to values for overloaded symbols.

For example, the \textit{Collection} class becomes a single parameter class with an associated type definition:

```haskell
class Collection \( E \) {
  type \( C : * \to * \)
  find : \( C \to E \to Option E \)
  add : \( C \to E \to C \) 
}

inst Collection Eq \( X \Rightarrow Collection \) \( X \) \{type \( C = List E \), ... \}

inst Collection Eq \( X \Rightarrow Collection \) \( X \) \{type \( C = Set E \), ... \}
```

Associated types increase the expressiveness of type classes.

**Overlapping instances**  In practice, overlapping instances may be desired! This seems in contradiction with the fact that overlapping instances are a source of incoherence. For example, one could provide a generic implementation of sets provided an ordering relation on elements, but also provide a more efficient version for bit sets. When overlapping instances are allowed, further rules are needed to disambiguate the overlapping resolution and preserve coherence. For instance, priority rules may be used. An interesting resolution strategy is to give priority to the most specific match.

However, the semantics depend on some particular resolution strategy and becomes more fragile. See [Jones et al. (1997)](1997) for a discussion. See also [Morris and Jones (2010)](2010) for a recent new proposal. For example, the definitions:

```haskell
inst Eq(\( X \)) \{ equal = (=) \}
inst Eq(Int) \{ equal = (==) \}
```

could elaborate into the creation of both a generic dictionary and a specialized one.

```haskell
let Eq X : Eq X = \{ equal = (=) \}
let EqInt : Eq Int = \{ equal = (==) \}
```

Then, \( EqInt \) or \( Eq Int \) are two dictionaries of type \( Eq Int \) but with different implementations.
7.4. **VARIATIONS**

**Restriction that are harder to lift**  We have made several restrictions to the definition of type classes. Some can be lifted at the price of some tolerable complication. Relaxing other restrictions, even if it could make sense in theory, would raise serious difficulties in practice.

For example, allowing constrained type schemes of the form $K\tau$ instead of the restricted form $K\alpha$ would affect many aspects of the language and it would becomes much more difficult to control the termination of constrained resolution and of the elaboration of dictionaries.

Allowing class instances of the form $\text{inst}\ \forall\vec{\beta}.\ \vec{P} \Rightarrow K\tau\{\rho\}$ where $\tau$ is $G\vec{\tau}$ and not just $G\vec{\beta}$, it would become difficult to check non-overlapping of class instances.

**Implicit values**

Implicit values are a mechanism that allows to build values from types. The implies a way to populate an environment of definitions that can be used to build implicit values and a mechanism to introduce place holders where values should be build from their types.

Implicit values have been used in the language **Scala** for implicit conversions (but they can do more). An extension of OCaml with implicit values is being prototyped. Implicit values have also been proposed as an alternative to **Haskell** type classes (Oliveira et al. 2012).

**Conclusions**

**Methods as overloading functions**  One approach to object-orientation is to see methods as overloaded functions. Then, objects carry class tags that can be used at runtime to find the best matching definition. This approach has been studied in detail by Millstein and Chambers (1999). See also Bonniot (2002, 2005).

**Summary**  Static overloading is not a solution for polymorphic languages. Dynamics overloading must be used instead. The implementation of type classes in the **Haskell** language has proved quite effective: it is a practical, general, and powerful solution to dynamic overloading. Moreover, it works relatively well in combination with ML-like type inference.

However, besides the simplest case of overloading on which everyone agrees, some useful extensions often come with serious drawbacks, and they are not yet an agreement on the best design compromises. In **Haskell**, the design decisions have often been in favor of expressiveness, but then losing some of the properties and the canonicity of the minimalistic initial design.

Dynamic overloading is a typical and very elegant use of elaboration. The programmer could in principle write the elaborated program manually, explicitly building and passing
dictionaries around, but this would be cumbersome, tricky, error prone, and it would sign-
ificantly obfuscate the code. Instead, the elaboration mechanism does this automatically,
without arbitrary choices (in the minimal design) and with only local transformations that
preserve the structure of the source program.

**Further reading**  For an all-in-one explanation of Haskell-like overloading, see *The essence of Haskell* by Odersky et al. See also the Jones’s monograph *Qualified types: theory and practice*. For a calculus of overloading see the ML& calculus proposed by Castagna (1997).

Recently, type classes have also been added to Coq Sozeau and Oury (2008). Interestingly, the elaboration of proof terms need not be coherent which makes it a simpler situation for overloading.
7.5  OMMITTED PROOFS AND ANSWERS TO EXERCISES

7.5  Ommitted proofs and answers to exercises

Solution of Exercise 42

We first need to show that the $\delta_3$ preserves typings. Assume that

$$\Gamma \vdash \text{unpack}_{\exists \alpha. \tau} (\text{pack}_{\exists \alpha. \tau} \tau' V) : \tau_0$$

By inversion of typing, $\tau_1$ and $\tau_0$ must be equal to $\tau$ and $\forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta$, respectively, and the judgment $\Gamma \vdash V : [\alpha \mapsto \tau']$ must hold. Let $\Gamma'$ be $\Gamma, \beta, y : \forall \alpha. \tau \rightarrow \beta$. By weakening, we have $\Gamma' \vdash V : [\alpha \mapsto \tau']$. We then have $\Gamma' \vdash y \tau' V : \beta$ and finally, we have

$$\Gamma \vdash \Lambda \beta. \lambda y : \forall \alpha. \tau \rightarrow \beta. y \tau' V : \tau_0$$

as expected.

We then need to show that $\delta_3$ satisfies progress, i.e., a full well typed application of $\text{unpack}_{\exists \alpha. \tau}$ can always be reduced. Assume that $\Gamma \vdash \text{unpack}_{\exists \alpha. \tau} V : \tau_0$. By inversion of typing, we must have $\Gamma \vdash V : \exists \alpha. \tau$. By the classification lemma (to be extended and rechecked), $V$ must be an existential value, i.e., of the form $\text{pack}_{\exists \alpha. \tau} \tau_0 V_0$. Hence, $\text{unpack}_{\exists \alpha. \tau} V$ reduces by $\delta_3$.

Solution of Exercise 43

We just force $\tau_1$ to coincide with $\tau$:

$$\text{unpack}_{\exists \alpha. \tau} (\text{pack}_{\exists \alpha. \tau} \tau' V) \rightarrow \Lambda \beta. \lambda y : \forall \alpha. \tau \rightarrow \beta. y \tau' V$$

(\delta_3)

The proof of subject reduction will know by construction that $\tau_0$ is $\tau$ instead of learning it by inversion of typing. Conversely for progress, we will have to show that $\tau_1$ and $\tau$ are equal by inversion so that $\delta_3$ can be applied.

Solution of Exercise 45

Let $M_1$ be if $M$ then $V_1$ else $V_2$ where $V_i$ is of the form $\text{pack} \tau_i, V_i$ as $\exists \alpha \tau$ and the two witnesses $\tau_1$ and $\tau_2$ differ. There is no common type for the unpacking of the two possible results $V_1$ and $V_2$. The choice between those two possible results must be made, by evaluating $M_1$, before unpacking.

Solution of Exercise 47

The answer is in the 2007–2008 exam.
Bibliography

A tour of scala: Implicit parameters. Part of scala documentation.


