Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of September 20, 2017
## 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 ............................................... 25
    3.4.1 Proof of subject reduction ............................ 26
    3.4.2 Proof of progress ...................................... 28
  3.5 Simple extensions ............................................ 30
    3.5.1 Unit ...................................................... 30
    3.5.2 Boolean .................................................. 30
    3.5.3 Pairs ...................................................... 31
    3.5.4 Sums ....................................................... 32
    3.5.5 Modularity of extensions ............................... 32
    3.5.6 Recursive functions ..................................... 33
    3.5.7 A derived construct: let-bindings .................... 33
  3.6 Exceptions ................................................... 35
    3.6.1 Semantics ................................................. 35
## CONTENTS

3.6.2 Typing rules ........................................... 36  
3.6.3 Variations ............................................. 37  
3.7 References .................................................. 39  
3.7.1 Language definition ................................... 39  
3.7.2 Type soundness ....................................... 41  
3.7.3 Tracing effects with a monad ......................... 42  
3.7.4 Memory deallocation ................................... 43  
3.8 Ommitted proofs and answers to exercises .......... 44  

4 Polymorphism and System F ........................................ 49  
4.1 Polymorphism ............................................. 49  
4.2 Polymorphic λ-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 .... 69  
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 λ-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 J ................................. 98  
5.3.2 Constraints ........................................... 99
CONTENTS

5.3.3 Constraint solving by example ............................................ 103
5.3.4 Type reconstruction .......................................................... 106
5.4 Type annotations ................................................................. 109
  5.4.1 Explicit binding of type variables ....................................... 110
  5.4.2 Polymorphic recursion ...................................................... 113
  5.4.3 mixed-prefix ................................................................. 114
5.5 Equi- and iso-recursive types ................................................ 115
  5.5.1 Equi-recursive types ....................................................... 115
  5.5.2 Iso-recursive types ....................................................... 117
  5.5.3 Algebraic data types ...................................................... 118
5.6HM(X) .................................................................................. 119
5.7 Type reconstruction in System F .............................................. 121
  5.7.1 Type inference based on Second-order unification .................. 121
  5.7.2 Bidirectional type inference .............................................. 122
  5.7.3 Partial type inference in MLF ............................................ 124
5.8 Proofs and Solution to Exercises ............................................ 124

6 Existential types ................................................................. 127
  6.1 Towards typed closure conversion .......................................... 128
  6.2 Existential types .................................................................. 130
    6.2.1 Existential types in Church style (explicitly typed) .............. 130
    6.2.2 Implicitly-typed existential types .................................... 133
    6.2.3 Existential types in ML .................................................. 135
    6.2.4 Existential types in OCaml ............................................ 136
  6.3 Typed closure conversion ..................................................... 137
    6.3.1 Environment-passing closure conversion ............................ 137
    6.3.2 Closure-passing closure conversion .................................. 139
    6.3.3 Mutually recursive functions ......................................... 141

7 Overloading ................................................................. 145
  7.1 An overview ...................................................................... 145
    7.1.1 Why use overloading? ..................................................... 145
    7.1.2 Different forms of overloading ........................................ 146
    7.1.3 Static overloading ......................................................... 147
    7.1.4 Dynamic resolution with a type passing semantics .............. 147
    7.1.5 Dynamic overloading with a type erasing semantics .......... 148
  7.2 Mini Haskell ....................................................................... 149
    7.2.1 Examples in MH ............................................................ 149
    7.2.2 The definition of Mini Haskell ........................................ 150
    7.2.3 Semantics of Mini Haskell ............................................. 152
    7.2.4 Elaboration of expressions ............................................ 154
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:

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

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
\[ \forall \alpha_1 \alpha_2. (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_0 \rightarrow \alpha_1) \rightarrow \alpha_0 \rightarrow \alpha_2 \]
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
\[ \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list} \alpha \rightarrow \text{list} \alpha \]
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 \( \forall \alpha. \alpha \rightarrow \alpha \) has only one inhabitant, namely the identity. Similarly, the type of the list sorting function
\[ \forall \alpha. (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{list} \alpha \rightarrow \text{list} \alpha \]
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 \( \forall \alpha. \alpha \rightarrow \alpha \).

  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 \( \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \).

- 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. \( \lambda x. \text{show} (\text{read} x) \)).
4.2. POLYMORPHIC $\lambda$-CALCULUS

![Typing rules for System F.]

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:

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

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:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha$$
We extend our previous convention to form environments: \( \Gamma, \alpha \) extends \( \Gamma \) with a new variable \( \alpha \), provided \( \alpha \not\in \Gamma \), 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):

\[
\begin{align*}
\text{WfEnvEmpty} & \quad \Gamma \vdash \emptyset \\
\text{WfEnvVar} & \quad \Gamma \vdash x \not\in \text{dom}(\Gamma) \quad \Gamma \vdash \tau \\
\text{WfEnvTvar} & \quad \Gamma \vdash \alpha \not\in \text{dom}(\Gamma) \\
\text{WfTypeVar} & \quad \Gamma \vdash \alpha \in \text{dom}(\Gamma) \\
\text{WfTypeArrow} & \quad \Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2 \\
\text{WfTypeForall} & \quad \Gamma, \alpha \vdash \tau
\end{align*}
\]

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:

\[
\begin{align*}
\text{Var} & \quad x : \tau \in \Gamma \quad \Gamma \vdash \tau \\
\text{Tapp} & \quad \Gamma \vdash M : \forall \alpha.\tau \\
& \quad \Gamma \vdash \tau' \\
& \quad \Gamma \vdash M \ \tau' : [\alpha \mapsto \tau']\tau
\end{align*}
\]

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:

\[
(\Lambda \alpha. M) \tau \rightarrow [\alpha \mapsto \tau]M \quad (\ell)
\]

Then, there is a choice regarding whether type abstraction should stop the evaluation, or let reduction proceed.
4.2. POLYMORPHIC $\lambda$-CALCULUS

**Type-passing semantics** In most presentations of System $F$, type abstraction blocks the evaluation and is defined as follows:

$$E ::= [] M | V [] | [] \tau$$

$$V ::= \lambda x : \tau. M | \Lambda \alpha. M$$

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 $(f \ \text{int}) (\circ \ \text{int} (\text{list} \ \text{int}) (\text{list} (\text{list} \ \text{int}))) (f (\text{list} \ \text{int}))$ reduces to $\lambda x : \text{int}. f \ \text{int} (f (\text{list} \ \text{int}) \ x)$, while its more general polymorphic version

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

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 (Morrisett 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:

$$E ::= [] M | V [] | [] \tau | \Lambda \alpha. []$$

$$V ::= \lambda x : \tau. M | \Lambda \alpha. V$$

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

$$(\Lambda \alpha. V) \ \tau \rightarrow [\alpha \mapsto \tau]V \quad (\iota_v)$$

We now have:

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

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:

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

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.
4.2. **POLYMORPHIC \( \lambda \)-CALCULUS**

**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:

\[
\tau ::= \alpha | \tau \rightarrow \tau | \forall \alpha. \tau | \tau \times \tau
\]

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:

\[
V ::= \ldots | \text{Pair} | \text{Pair} \tau | \text{Pair} \tau \tau | \text{Pair} \tau \tau V | \text{Pair} \tau \tau V V |
\text{proj}_1 | \text{proj}_1 \tau | \text{proj}_1 \tau \tau
\]

We add the two following reduction rules:

\[
\text{proj}_i \tau_1 \tau_2 (\text{Pair} \tau'_1 \tau'_2 \tau_1 \tau_2) \rightarrow V_i \quad (\delta_{\text{pair}})
\]

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:

\[
\text{proj}_i \tau_1 \tau_2 (\text{Pair} \tau_1 \tau_2 \tau_1 \tau_2) \rightarrow V_i \quad (\delta'_{\text{pair}})
\]

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 \( \text{arity} (c) \). Constants must actually be partitioned into constructors (written \( C \)) and destructors (written \( d \)); moreover, we disallow nullary destructors\(^1\).

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:

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

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:

\[
d \ \tau_1 \ldots \tau_j \ V_1 \ldots \ V_k \rightarrow M \quad (\delta_d)
\]

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

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

We assume that types respect the arities of type constructors, \( i.e. \ n \) is equal to \( \text{arity} (G) \) 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 \( \forall \alpha_1 \ldots \forall \alpha_j \cdot \tau_1 \rightarrow \ldots \tau_n \rightarrow \tau \). 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 \( \Delta \):

\[
\begin{align*}
& \text{Cst} \\
& \quad c : \tau \in \Delta \\
& \quad \vdash \Gamma \\
& \Gamma \vdash c : \tau
\end{align*}
\]

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 \( \forall \alpha_1 \ldots \alpha_p \cdot \tau'_1 \rightarrow \ldots \tau'_q \rightarrow \tau \), then values will contain:

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

\(^1\)Nullary polymorphic destructors introduce pathological cases to maintain the semantics type-erasing—for little benefit in return.
4.2. POLYMORPHIC λ-CALCULUS

and

\[ c \mid c \tau_1 \mid \ldots \mid c \tau_1 \ldots \tau_p \mid c \tau_1 \ldots \tau_p V_1 \mid \ldots \mid c \tau_1 \ldots \tau_p V_1 \ldots V_{q-1} \]

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

**Definition 1** δ-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 δ-rules is sufficient to ensure soundness of the extension.

For example, to add a unit constant, we only introduce a type constant \text{unit} and a constructor \( () \) of arity 0 of type \text{unit}. As no primitive is added, δ-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 δ-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. 2) Check that this extension is sound. (Solution p. 84)

**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:

\[ \text{fix} : \forall \alpha. \forall \beta. ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \in \Delta \]

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 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 (Recursion with datatypes) In ML a one-constructor datatype can be used to emulate recursive types, namely a type Any such that a value of type any → any can be converted to a value of type any, and conversely. Give the definition in ML. Describe the extension as the addition of new constants. Verify the soundness of δ-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 let rec or implicit recursive types (the −rectypes option). (See Exercise 7 for a definition of the fix-point in the λ-calculus or in ML with recursive types.)

(Solution p. 85)

4.3 Type soundness

We prove type soundness for System F with constants, assuming the soundness of δ-reduction.

The structure of the proof is similar to the case of simply-typed λ-calculus and follows from subject reduction and progress. Subject reduction uses the following auxiliary lemmas: inversion of typing rules (Lemma 13), permutation (Lemma 14), weakening (Lemma 15), expression substitution (Lemma 16), type substitution (Lemma 17), and compositionality of typing (Lemma 18).

Lemma 13 (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 14 (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 15 (Weakening) If \( \Gamma \vdash M : \tau \) and \( \Gamma, \Gamma' \vdash M : \tau \).
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 16 (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 17 (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. (Details of the proof p. 86)

Exercise 29 Write the details of the proof.

Lemma 18 (Compositionality) If $\Gamma \vdash E[M] : \tau$, then there exists a sequence of type variables $\bar{\alpha}$ 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.

Notice that $M'$ is typechecked in the context $\Gamma$ extended with $\bar{\alpha}$, since the hole in the context $E$ may be under type abstractions. We use the notation $\bar{\alpha}$ for a (possibly empty) sequence of 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*}
\text{Abs} & \quad \Gamma, x : \tau' \vdash M_1 : \tau \\
\text{App} & \quad \Gamma \vdash (\lambda x: \tau'. M_1) : \tau' \rightarrow \tau \\
& \quad \Gamma \vdash V : \tau' \\
\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).
4.3. **TYPE SOUNDNESS**

The classification lemma, which is a key to progress, is slightly modified to account for polymorphic types and constructed types. We need to state the lemma under an arbitrary set of type variables \( \vec{\alpha} \) instead of the empty context—because evaluation is allowed under type abstractions.

**Lemma 19 (Classification)** Assume \( \vec{\alpha} \vdash V : \tau \)

- 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 \( \vec{\alpha} \vdash M : \tau \) and \( M \rightarrow \), then \( M \) is some value \( V \).

The theorem has been strengthened, using a sequence of type variables \( \vec{\alpha} \) 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 \( \vec{\alpha} \vdash M : \tau \) and \( M \) is irreducible.

*Case \( M \) is \( x \):* This is not possible since \( x \) is not well-typed in \( \vec{\alpha} \).

*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, \( \vec{\alpha} \vdash M_1 : \tau_2 \rightarrow \tau_1 \) and \( \vec{\alpha} \vdash M_2 : \tau_2 \). 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 irreducible. Hence, by induction hypothesis, \( M_2 \) is also a value. Since the application \( M_1 M_2 \) itself cannot be reduced, \( M_1 \) is not a function. Since it has an arrow type, it follows from the classification lemma that it 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.
CHAPTER 4. POLYMORPHISM AND SYSTEM F

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 $[F]$ 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 $\text{ir-}$ 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 $\alpha \not\in \Gamma$ due to the assumption on the formation of typing environments.
### 4.4. Type Erasing Semantics

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if-VAR</strong></td>
<td>$\Gamma \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td><strong>if-Cst</strong></td>
<td>$\Gamma \vdash c : \Delta(c)$</td>
</tr>
</tbody>
</table>
| **if-Abs** | $\Gamma, x : \tau_0 \vdash a : \tau$   \[
\Gamma \vdash \lambda x. a : \tau_0 \rightarrow \tau\] |
| **if-App** | $\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1$ \[\Gamma \vdash a_2 : \tau_1\] |
| **if-Tabs** | $\Gamma, \alpha \vdash a : \tau$ \[\Gamma \vdash a : \forall \alpha. \tau\] |
| **if-Tapp** | $\Gamma \vdash a : \forall \alpha. \tau$ \[\Gamma \vdash a : [\alpha \mapsto \tau]_0\] |

#### Figure 4.2: Typing rules for explicitly-typed System F.

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if-Tabs-Bis</strong></td>
<td>$\Gamma \vdash a : \tau$ [\alpha # \Gamma] [\Gamma \vdash a : \forall \alpha. \tau]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if-VAR</strong></td>
<td>$x : \alpha_1 \vdash x : \alpha_1$</td>
</tr>
<tr>
<td><strong>Broken if-Tabs</strong></td>
<td>$x : \alpha_1 \vdash x : \forall \alpha_1, \alpha_1$</td>
</tr>
<tr>
<td><strong>if-Tapp</strong></td>
<td>$x : \alpha_1 \vdash x : \forall \alpha_2, \alpha_2$</td>
</tr>
<tr>
<td><strong>if-Abs</strong></td>
<td>$\emptyset \vdash \lambda x. x : \alpha_1 \rightarrow \alpha_2$</td>
</tr>
<tr>
<td><strong>Broken Tabs</strong></td>
<td>$x : \alpha_1 \vdash x : \forall \alpha_1, \alpha_1$</td>
</tr>
<tr>
<td><strong>if-Tapp</strong></td>
<td>$x : \alpha_1 \vdash x : \forall \alpha_2, \alpha_2$</td>
</tr>
<tr>
<td><strong>if-Abs</strong></td>
<td>$\emptyset \vdash \lambda x : \forall \alpha_1, \forall \alpha_2, \alpha_1 \rightarrow \alpha_2$</td>
</tr>
</tbody>
</table>

A good intuition is that a judgment $\Gamma \vdash a : \tau$ corresponds to the logical assertion $\forall \alpha. (\Gamma \Rightarrow (a : \tau))$, where $\alpha$ are the free type variables of the judgment, taken in any order. In this view, `Tabs-Bis` corresponds to the axiom:

$$\forall \alpha. (P \Rightarrow Q) \equiv P \Rightarrow (\forall \alpha. Q)$$

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.

**Lemma 20** $\Gamma \vdash a : \tau$ 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 \( F \) 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 (Distrib pair, disjoint types)** Find two terms \( a_1 \) and \( a_2 \) such that \( a_1 \) has type \( \tau_1 \) 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 (Distrib pair in \( F^\omega \))** Only if you know System \( F^\omega \): find a type \( \tau_0 \) for \( a_0 \) in System \( F^\omega \) that is more general than both \( \tau_1 \) and \( \tau_2 \), i.e. from which \( \tau_1 \) and \( \tau_2 \) can be obtained by rule [Inst-Gen].

(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:

\[
\text{Inst-Gen} \quad \beta \neq \forall \alpha. \tau \\
\forall \alpha. \tau \leq \forall \beta. [\alpha \mapsto \tau] \tau
\]
Notice that \( \vec{\alpha} \) and \( \vec{\beta} \) stands for (possibly empty) sequences of type variables. 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\(^2\) in the implicitly-typed version:

\[
\frac{\Gamma \vdash a : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash a : \tau_2}
\]

Perhaps surprisingly, the rule is not derivable\(^3\) in our presentation of System F. Although, we have the following derivation,

\[
\frac{\Gamma, \vec{\beta} \vdash a : \forall \vec{\alpha}. \tau}{\Gamma \vdash \lambda \vec{\beta}. M \, \vec{\tau} : \forall \vec{\beta}. [\vec{\alpha} \mapsto \vec{\tau}] \tau}
\]

the premise \( \Gamma, \vec{\beta} \vdash a : \forall \vec{\alpha}. \tau \) can only be justified from the assumption \( \Gamma \vdash a : \forall \vec{\alpha}. \tau \) by an application of weakening (the side condition \( \vec{\beta} \# \forall \vec{\alpha}. \tau \) of rule \textsc{Gen} ensures that \( \Gamma, \vec{\beta} \) is well-formed.) Otherwise, in context \( \Gamma \) alone, \( \vec{\tau} \) would not necessarily be well-formed, as required by rule \textsc{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 \textsc{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 counter-part 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 \( \forall \vec{\alpha}. \tau \) and \( \forall \vec{\beta}. [\vec{\alpha} \mapsto \vec{\tau}] \tau \) where \( \vec{\beta} \# \forall \vec{\alpha}. \tau \). \textit{W.l.o.g.}, we may assume that \( \vec{\beta} \# \Gamma \) (6), as it may always be satisfied up to a renaming of bound variables \( \vec{\beta} \). Then, we have the pseudo-derivation on the left-hand side (where the weakening lemma is used as a pseudo-typing rule \textsc{Weakening}), which can be abbreviated by the admissible typing rule \textsc{Sub} given on the right-hand side.

\[
\frac{\Gamma \vdash M : \forall \vec{\alpha}. \tau}{\overset{\text{Weakening}}{\frac{\vec{\beta} \# \forall \vec{\alpha}. \tau}{\vec{\beta} \# \Gamma}}}
\]

Admissible rule:

\[
\frac{\Gamma \vdash M : \forall \vec{\alpha}. \tau \quad \vec{\beta} \# \forall \vec{\alpha}. \tau}{\Gamma \vdash \lambda \vec{\beta}. M \, \vec{\tau} : \forall \vec{\beta}. [\vec{\alpha} \mapsto \vec{\tau}] \tau}
\]

\(^2\)A rule is \textit{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 \textit{derivable} if it can be replaced by a sub-derivation tree with the same premises and conclusion.
In F, we rather write subtyping as a judgment \( \Gamma \vdash \tau_1 \leq \tau_2 \) instead of the binary relation \( \tau_1 \leq \tau_2 \) 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 \( \Lambda \vec{\beta}.[] \vec{\tau} \). 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 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:

\[
\mathcal{R} ::= [] \mid \Lambda \alpha. \mathcal{R} \mid \mathcal{R} \vec{\tau}
\]

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

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 \( \mathcal{R} \) such that \( \Gamma \vdash \mathcal{R}[\tau_1] : \tau_2 \).

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 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:

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

With this larger instance relation, Rule \textbf{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 F_\( \eta \), since it is also the closure of System F by \( \eta \)-expansion; that is, a term is in System F_\( \eta \) if and only if it is the \( \eta \)-conversion of a term in System F.
Exercise 32 1) Show that \(\forall \alpha. \tau \equiv \tau\) whenever \(\alpha \notin \text{ftv}(\tau)\). 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. 87)

One may wonder what System \(F_\eta\) brings to System \(F\) that it does not already have. Consider the identity function \(\text{id}\) 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:

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

The type \(\forall \alpha. \alpha \rightarrow \alpha\) is actually a principal type for \(\text{id}\) 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 \(ch\) of \(ch\) to \(id\). How can it be typed? If we keep \(id\) polymorphic, then \(ch\) has type \((\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)\), say \(\tau_1\); if, on the opposite, we instantiate \(id\), then \(ch\) has type \(\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)\), say \(\tau_2\)—as in \(\text{ML}\) 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 \(ch\) 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 \(ch\) 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 \(\text{MLF}\) has been design 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. TYPE ERASING SEMANTICS

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-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 $a_1$ of type $τ$ in $⌈F⌉$ that reduces to $a_2$, can we find a term $M_1$ in $F$ whose erasure is $a_1$ and show that $M_1$ reduces to a term $M_2$ whose erasure is $a_2$ to conclude that the type of $a_2$ is the type of $a_1$? 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 direct and a inverse simulation.

**Lemma 21 (Direct simulation)** The reduction in $F$ is simulated in $⌈F⌉$ up to $ι$-steps. Assume $Γ ⊢ M : τ$. Then:
1) If $M ↠_ι M′$, then $⌈M⌉ = ⌈M′⌉$
2) If $M ↠_βδ M′$, then $⌈M⌉ ↠_βδ ⌈M′⌉$

The inverse 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 22 (Inverse simulation)** Assume $Γ ⊢ M : τ$ and $⌈M⌉ ↠_ι a$. Then, there exists a term $M′$ such that $M ↠_ι ↠_βδ M′$ 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:

**(Direct bisimulation)** If $M \rightarrow_\delta M'$, then $\lceil M \rceil \rightarrow_\delta \lceil M' \rceil$.

**(Inverse bisimulation)** If $\lceil M \rceil \rightarrow_\delta a$, 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 $\mathcal{F}$ with pairs of the exercise 24 (p. 77). 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.

(Solution p. 87)

The direct simulation (Lemma 21) is straightforward to establish. (Details of the proof p. 88)

The inverse 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 23**

1) A term that erases to $\bar{e}[a]$, then $M_0$ is of the form $\bar{E}[M]$ where $\lceil \bar{E} \rceil$ is $\bar{e}$ and $\lceil M \rceil$ 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 $\bar{R}$.

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

The main helper lemma is:

**Lemma 24 (Inversion of type erasure)** Assume $\lceil M \rceil = a$

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

The proof is by an induction on $M$. 

4.4. TYPE ERASING SEMANTICS

Lemma 25 (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 \( R[c \bar{\tau} V_1 \ldots V_n] \) 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 26 Let \( M \) be a well-typed term in \( \iota \)-normal whose erasure is \( a \)

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

We may now prove inverse 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 \( F \). Its direct proof is also immediate, as \( \iota \)-reduction strictly decreases the number of type abstractions.

---

Proof (inverse 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 26 \( M \) 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 23 \( M \) is of the form \( E[M_1] \) where \([E] = \tilde{e}\) and \([M_1] = a_1\). By compositionality (Lemma 18), \( M_1 \) 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 \( \tilde{e}[a_2] \), i.e. \( a \).

Case \([M]\) is a full application \((d \ v_1 \ldots v_n)\) and reduces to \( a \): By Corollary 26 \( M \) is of the form \( R[M_0] \) where \( M_0 = d \bar{\tau} V_1 \ldots V_n \) a \([V_i] = v_i\). Since \([M_0] \rightarrow a \), by the inverse assumption for \( \delta \)-rules, there exists \( M_0' \) such that \( M_0 \rightarrow_{\delta} M_0' \) and \([M_0'] = a \). Let \( M' \) be \( R[M_0'] \). Since \( R \) is an evaluation context, we have \( M \rightarrow_{\delta} M' \) and \([M'] = a \).

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

Progress and subject reduction holds in implicitly-typed System F.

Proof: Assume that $\Gamma \vdash a_1 : \tau$. By Lemma 20, there exists a term $M_1$ such that $\Gamma \vdash M_1 : \tau$ and $\lceil M_1 \rceil$ is $a_1$.

Progress: Let $M_2$ be the $\iota$-normal form of $M_1$. By direct simulation, $\lceil M_2 \rceil$ is $a$. By subject reduction, we have $\Gamma \vdash M_2 : \tau$. By progress in $F$, either $M_2$ $\beta\delta$-reduces and so does $a$, by direct simulation (Lemma 21) or $M_2$ is a value and so is its erasure $a_1$ (by observation).

Subject reduction: Assume $a_1 \rightarrow a_2$. By inverse simulation (Lemma 22), there exists a term $M_2$ such that $M_1 \rightarrow^* \iota\rightarrow^* \beta\delta M_2$ and $\lceil M_2 \rceil$ is $a_2$. By subject reduction in $F$, we have $\Gamma \vdash M_2 : \tau$. By Lemma 20, we have $\Gamma \vdash a_2 : \tau$, 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 direct simulation is usually straightforward, the inverse 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
4.5. REFERENCES

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:

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

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

\[
\begin{align*}
\text{ref } \tau V / \emptyset \rightarrow \ell / \ell \mapsto V \\
\Lambda \alpha. \text{ref } \tau V / \emptyset \rightarrow \Lambda \alpha. \ell / \ell \mapsto V
\end{align*}
\]

While we have

\[
\alpha \vdash \text{ref } \tau V / \emptyset : \text{ref } \tau \\
\text{and} \\
\alpha \vdash \ell / \ell \mapsto V : \text{ref } \tau
\]

We have

\[
\vdash \Lambda \alpha. \text{ref } \tau V / \emptyset : \forall \alpha. \text{ref } \tau \\
\text{but not} \\
\vdash \Lambda \alpha. \ell / \ell \mapsto V : \forall \alpha. \text{ref } \tau
\]

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 [Loc] given below, page [4.5.2]):

\[
\begin{align*}
\Delta \vdash \ell : \text{ref } \tau \\
\Sigma \vdash \Lambda \alpha. \ell : \forall \alpha. \text{ref } \tau
\end{align*}
\]

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 [Tabs] cannot be applied.

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

\[
\begin{align*}
\alpha \vdash \text{ref } \tau V : \text{ref } \tau \\
\emptyset \vdash \Lambda \alpha. \text{ref } \tau V : \forall \alpha. \text{ref } \tau
\end{align*}
\]
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, $\alpha \to \alpha$ 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.

### Value forms

- $U :: x \mid V \mid \Lambda \alpha. U \mid U \tau$

The problematic proof case *vanishes*, as we now never $\beta\delta$-reduce under type abstraction, only $\iota$-reduction is possible. Subject reduction holds again. Let us prove it.

#### 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) $\text{let rec } m : \Sigma = \mu \text{ in } [\ell \mapsto m.\ell] \cdot M$ 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:

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

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:

- **CONFIG**
  
  $\bar{\alpha} \vdash M : \tau \\
  \bar{\alpha} \vdash \mu : \Sigma \\
  \bar{\alpha} \vdash M / \mu : \tau$

- **STORE**
  
  $\forall \ell \in \text{dom}(\mu), \bar{\alpha}, \Sigma, \emptyset \vdash \mu(\ell) : \Sigma(\ell) \\
  \bar{\alpha} \vdash \mu : \Sigma$

Closed configurations must be typed in an environment composed of type variables. No new type variables is never introduced during reduction. These type variables may appear in the store typing during reduction, there are thus placed in front the store typing and cannot be generalized.
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:

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

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:

\[
\text{ref} \, \tau \, V / \mu \rightarrow \ell / \mu[\ell \mapsto V] \quad \text{if } \ell \notin \text{dom}(\mu)
\]

\[
\ell := (\tau) \, V / \mu \rightarrow () / \mu[\ell \mapsto V] \quad !\tau \, \ell / \mu \rightarrow \mu(\ell) / \mu
\]

Type soundness can now be stated as

**Lemma 27** \( \delta \)-rules preserve well-typedness of closed configurations.

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

**Lemma 28** 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:

\[
\begin{align*}
\text{let } y : \forall \alpha. \text{ref} \, (\alpha \to \alpha) = \Lambda \alpha. \text{ref} \, (\alpha \to \alpha) \, (\lambda z : \alpha. z) \text{ in } \\
(y \, \text{bool}) := (\text{bool} \to \text{bool}) \, \text{not}; \\
!(\text{int} \to \text{int}) \, (y \, (\text{int})) \text{1}
\end{align*}
\]

Indeed, \( \text{ref} \, (\alpha \to \alpha) \, (\lambda z : \alpha. z) \) is not a value, but the application of a unary destructor to a value, so the expression \( \Lambda \alpha. \text{ref} \, (\alpha \to \alpha) \, (\lambda z : \alpha. z) \) 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 \( \forall \alpha. \text{list} \alpha \to \text{list} \alpha \) and \( \forall \alpha. \alpha \to \alpha \), the expression \( \Lambda \alpha. \text{map} \alpha (\text{id} \alpha) \) is now ill-typed. A common work-around is to perform a manual \( \eta \)-expansion \( \Lambda \alpha. \lambda y : \text{list} \alpha. \text{map} \alpha (\text{id} \alpha) \ y \). 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

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

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 \textsuperscript{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 \texttt{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: \textit{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 (\textit{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 \textit{not practical}, because they have \textit{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 \textit{practical}, because it leads to an efficient type inference algorithm (presented in chapter \S5); and it supports separate compilation.

4.6.1 Definition

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

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

The \texttt{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 \textit{types} and \textit{type schemes}. The syntax of \textit{types} is that of simply-typed $\lambda$-calculus, but a separate category of \textit{type schemes} is introduced:

\[
\tau ::= \alpha \mid \tau \rightarrow \tau \mid \ldots \quad \sigma ::= \tau \mid \forall \alpha. \sigma
\]
All quantifiers must appear in \textit{prenex position}, so type schemes are less expressive than System-F types. We often write $\forall \alpha . \tau$ as a short hand for $\forall \alpha_1 . . . \forall \alpha_n . \tau$. When viewed as a subset of System F, one must think of \textit{type schemes} are the primary notion of types, of which \textit{types} are a subset.

An ML typing context $\Gamma$ binds program variables to \textit{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 \texttt{Let} moves a type scheme into the environment, which \texttt{Var} can exploit. Rule \texttt{Abs} and \texttt{App} are unchanged. \textit{$\lambda$-bound variables receive a monotype}. Rule \texttt{Gen} and \texttt{Inst} are as in implicitly-typed System F, except that \textit{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 \texttt{IML-} prefix of typing rules):

\[
\begin{array}{c}
\texttt{Var} \\
\alpha, z : \alpha \vdash z : \alpha \\
\texttt{Abs} \\
\alpha \vdash \lambda z. z : \alpha \rightarrow \alpha \\
\texttt{Gen} \\
\emptyset \vdash \lambda z. z : \forall \alpha . \alpha \rightarrow \alpha \\
\texttt{Pair} \\
\emptyset \vdash \text{let } f = \lambda z. z \text{ in } (f\ 0, f\ \text{true}) : \text{int} \times \text{bool}
\end{array}
\]

Notice that Rule \texttt{Gen} is used above \texttt{Let} (on the left-hand side), and \texttt{Inst} is used below \texttt{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 \textit{“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 $\forall \alpha . \alpha \rightarrow \alpha$. 

Figure 4.3: Typing rules for ML
4.6. DAMAS AND MILNER’S TYPE SYSTEM

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 following syntax:

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

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 \( \vdash_{F} \) to ML environments 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 $x\mathsf{ML}$ be the subset of explicitly-typed $\mathsf{ML}$ defined by the following grammar

\[
N \in x\mathsf{ML} \quad ::= \quad \Lambda \vec{\alpha}.Q \\
Q \quad ::= \quad x\vec{\tau} \mid Q \ Q \mid \lambda x:\tau.Q \mid \text{let } x:\sigma = N \text{ in } Q
\]

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

Specializing the typing rules of $e\mathsf{ML}$ (Figure 4.4) to the syntax of $x\mathsf{ML}$ on Figure 4.5 gives the typing rules of $x\mathsf{ML}$ on Figure 4.6. By construction, terms of $x\mathsf{ML}$ are a syntactic subset of terms of $e\mathsf{ML}$. By construction, we also have if $\Gamma \vdash_{x\mathsf{ML}} M : \sigma$ then $\Gamma \vdash_{e\mathsf{ML}} M : \sigma$.

Conversely, we wish to show that any term $M$ typable in $e\mathsf{ML}$ can be mapped to a term $N$ typable in $x\mathsf{ML}$ 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:

\[
\begin{align*}
\Gamma, \vec{\alpha} & \vdash Q : \tau & \Gamma, x : \tau_0 & \vdash Q : \tau & \Gamma \vdash Q_1 : \tau_2 \rightarrow \tau_1 & \Gamma \vdash Q_2 : \tau_2 \\
\Gamma \vdash \Lambda \vec{\alpha}.Q : \forall \vec{\alpha}.\tau & & \Gamma \vdash \lambda x:\tau_0.Q : \tau_0 \rightarrow \tau & & \Gamma \vdash Q_1 : \tau_2 \rightarrow \tau_1 & \Gamma \vdash Q_1 Q_2 : \tau_1 \\
\end{align*}
\]
4.6. DAMAS AND MILNER’S TYPE SYSTEM

- Type \( \eta \)-expansion of every occurrence of a variable according to the arity of its type scheme (Rule [VAR]). This ensures that every occurrence of a type variable will be fully specialized—hence assigned a monomorphic type.

- 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.

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

Moreover, if \( \Gamma \vdash M : \sigma \) holds, then \( \Gamma \vdash xML N : \sigma \) 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 [ Norm-Tapp], which performs strong \( \iota \)-reduction and uses type substitution (Lemma 17), and [ 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 [Inst] come immediately after [Var] and all applications of [Gen] come immediately above rule [Let] or at the bottom of the derivation—as imposed by the grammar of \( xML \) terms where \( Q \)-terms can only have monomorphic types.

In summary, any term of \( eML \) can be rearranged as a term of \( xML \) with the same type erasure. By dropping type information in terms of \( xML \), we then obtain a syntax-directed presentation of implicitly-typed \( ML \), called \( sML \):

Then, the judgments \( \Gamma \vdashML a : \tau \) and \( \Gamma \vdashSMLin a : \tau \) are equivalent.

However, for type inference, we rather use the equivalent presentation in Figure 4.7 called \( iML \) (or the inference type system) where type variables are not explicitly declared in the typing context—hence, the side condition for generalization on rule [LET].

In this final system, type substitution (Lemma 17), which we will use for type inference, can be restated as follows:

**Lemma 29 (Type Substitution)** Typings are stable by substitution.
If \( \Gamma \vdash a : \tau \) then \( \varphi \Gamma \vdash a : \varphi \tau \). for any substitution \( \varphi \).
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 eML The proof of subject reduction follows the same schema as for System $F$ (Theorem 10). 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 eML. 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 $\bar{\alpha} \vdash_{ML} a_1 : \sigma$ and $a_1 \rightarrow a_2$ hold. There exists an explicitly-typed term $M_1$ such that $\bar{\alpha} \vdash_{eML} M_1 : \sigma$ and $[M_1] = a_1$. By normalization, we may elaborate $M_1$ into a term $N_1$ of $eML$ such that $\bar{\alpha} \vdash_{eML} N_1 : \sigma$ and the $[N_1] = [M_1]$. Moreover, $N_1$ is by construction $\iota$-normal. Since $eML$ is a subset of System $F$, we have $\bar{\alpha} \vdash_F N_1 : \sigma$. By inverse simulation in System $F$ (Lemma 22), 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 $eML$ and such that $\bar{\alpha} \vdash_{eML} M_2 : \sigma$. Therefore, we have $\bar{\alpha} \vdash_{eML} M_2 : \sigma$ and since the type erasure of $M_2$ is that of $N_2$, i.e. $a_2$, we have $\bar{\alpha} \vdash_{ML} a_2 : \sigma$, as expected.

It thus remains to check that given a term $N_1$ such that $\Gamma \vdash_{xML} N_1 : \sigma$ and $N_1 \rightarrow_{\beta} N_2$, there exists a term $M_2$ in $xML$ that is a strong $\iota$-reduction of $N_2$ and such that $\Gamma \vdash_{xML} N_2 : \sigma$. 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:

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

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

$$\text{let } x : \forall \vec{\alpha}. \tau = V \text{ in } Q \mapsto [x \mapsto V]Q$$

then occurrences of $x$ in $Q$, which are of the form $x \vec{\tau}$, become $V \vec{\tau}$ and may contain $\iota$-redexes—which are not allowed in $xML$. Fortunately, $V$ is necessarily of the form $\Lambda \vec{\alpha}. V'$ where the arity of $\vec{\alpha}$ is equal to that of $\vec{\tau}$. Hence, we may immediately perform a sequence of $\iota$-reduction that brings the term back into $xML$ and in $\iota$-normal form. Notice however that this $\iota$-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 $\iota$-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 \(\Delta\):

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

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 \(\delta_i\) 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 \(\text{Pair } \tau_1 \tau_2 V_1 V_2\). 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 \(\forall \alpha. \text{bool} \to \alpha \to \alpha \to \alpha\) 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

1) 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 \(\forall \alpha. \alpha \to \text{list } \alpha \to \text{list } \alpha\); and one destructor \(\text{matchlist} \ldots\) of type:

\[
\forall \alpha \beta. \text{list } \alpha \to \beta \to (\alpha \to \text{list } \alpha \to \beta) \to \beta
\]
with the two reduction rules:

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

2) Ommitted.

**Solution of Exercise 27**

In ML, we may define the datatype:

```ml
type any = Fold of (any -> any)
```

This can be simulated by adding a new type `any`, a constructor `Any` and a destructor `unany` of types (any -> any) -> any and any -> (any -> any), respectively, with the following reduction rule:

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

Let us check soundness of this extension:

**Case Type preservation:** Assume that \( \Gamma \vdash \text{unfold } (\text{Fold } V) : \tau \). By inversion, we known that \( \tau \) is any -> 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 any -> any and \( \Gamma \vdash V : \text{any} \) holds. By classification, \( V \) must be Fold \( V_0 \). Hence, \( \text{unfold } V \) reduces.

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

```ml
let zfix g = (fun x -> x x) (fun z -> g (fun v -> z z v))
```

We may implement `zfix` in ML without recursive tyes as:

```ml
let unfold (Fold x) = x;
let zfix g =
  (fun x -> unfold (x (Fold x)))
  (fun z -> Fold (g (fun v -> unfold ((unfold z) z) v)))
```

**Proof of Lemma 16**

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 \text{VAR}.

**Case M is c:** By (1), \( c : \tau \) is in \( \Delta \). Hence, the goal (3) follows by Rule \text{VAR}.
Case $M$ is $\lambda y: \tau_1. M_1$: By (1), $\tau$ is of the form $\tau_2 \to \tau_1$ 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 $\mathbf{Abs}$ 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 $\mathbf{Tabs}$ 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 17**

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 $[\alpha \Rightarrow \tau]$. We must show $\Gamma, \theta \Gamma' \vdash \theta M : \theta \tau$ (2).

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 $\mathbf{Var}$.

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 (2) follows by Rule $\mathbf{Const}$.

Case $M$ is $\lambda x : \tau_0. M_1$: By (1) and inversion, we have $\Gamma, \alpha, \Gamma', x : \tau_0 \vdash M_1 : \tau_1$ where $\tau$ is $\tau_0 \to \tau_1$. By induction hypothesis, $\Gamma, \theta(\Gamma', x : \tau_0) \vdash M_1 : \tau_1$, i.e. $\Gamma, \theta \Gamma', x : \theta \tau_0 \vdash \theta M_1 : \theta \tau_1$. By Rule $\mathbf{Abs}$ we have $\Gamma, \theta \Gamma' \vdash \lambda x : \theta \tau_0. \theta M_1 : \theta \tau_0 \to \theta \tau_1$, i.e. (2).

Case $M$ is $\Lambda \beta. M_1$: By (1) and inversion, we have $\Gamma, \alpha, \Gamma', \beta \vdash M_1 : \tau_1$ where $\tau$ is $\forall \beta. \tau_1$. By induction hypothesis, we have $\Gamma, \theta(\Gamma', \beta) \vdash \theta M_1 : \theta \tau_1$, which is equal to $\Gamma, \theta \Gamma', \beta \vdash \theta M_1 : \theta \tau_1$. By rule $\mathbf{Tabs}$ we have $\Gamma, \theta \Gamma' \vdash \Lambda \beta. \theta M_1 : \forall \beta. \theta \tau$, i.e. i.e. (2).

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,

$$\Lambda \alpha_1. \Lambda \alpha_2. \Lambda \varphi_1. \Lambda \varphi_2. (\forall \alpha. \varphi_1(\alpha) \to \varphi_2(\alpha)) \to \varphi_1(\alpha_1) \to \varphi_1(\alpha_2) \to \varphi_2(\alpha_1) \times \varphi_2(\alpha_1)$$

for $\tau_0$. We recover $\tau_1$ by choosing the constant functions $\lambda \alpha. \alpha_1$ for $\varphi_1$ and $\tau_2$ by choosing the identity $\lambda \alpha. \alpha$ for both $\varphi_1$ and $\varphi_2$. \[\]
Solution of Exercise 32

1) Both directions follow from rule [INST-GEN] just applying the substitution $\alpha \mapsto \alpha$ for the direct implication and just generalizing over $\alpha$ for the reverse.

2) Rule [DISTR-RIGHT] is a particular case of [DISTRUTIVITY] indeed. Assuming $\alpha \notin \text{ftv}(\tau_1)$, and using the previous equivalence (1), we have

\[
\begin{align*}
\text{DISTRUTIVITY} & \quad (\forall \alpha. \tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2) \\
\text{CONGRUENCE-\rightarrow} & \quad \forall \alpha. (\tau_1 \rightarrow \tau_2) \leq \tau_1 \rightarrow (\forall \alpha. \tau_2)
\end{align*}
\]

Conversely, we have the following derivation:

\[
\begin{align*}
\text{INST-GEN} \quad \forall \alpha. \tau_1 \leq \tau_1 & \\
\text{CONGRUENCE-\rightarrow} & \quad \tau_1 \rightarrow \tau_2 \leq (\forall \alpha. \tau_1) \rightarrow \tau_2 \\
\text{CONGRUENCE-\leftarrow} & \quad \forall \alpha. (\tau_1 \rightarrow \tau_2) \leq \forall \alpha. (\forall \alpha. \tau_1) \rightarrow \tau_2 \\
\text{DISTR-RIGHT} & \quad \forall \alpha. (\tau_1 \rightarrow \tau_2) \leq (\forall \alpha. \tau_1) \rightarrow (\forall \alpha. \tau_2)
\end{align*}
\]

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:

\[
\text{proj}_i (\text{Pair} \ v_1 \ v_2) \rightarrow_{\delta} v_i
\]

The reduction $\text{proj}_1 (\text{Pair} \ v \ (\lambda x. \text{Pair} \ V)) \rightarrow_{\delta} v$ 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 (direct simulation); and it does not contains more (inverse 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 $[M]$ is $\text{proj}_i \ (\text{pair} \ [V_1] \ [V_2])$ which reduces to $[V_i]$ (2).

Case direct: Assume $M \rightarrow_{\delta} M'$. Then, $M'$ is $V_i$ and by (2), $[M] \rightarrow_{\delta} [V_i]$.

Case inverse: Assume that $[M] \rightarrow_{\delta} a$. Since reduction is deterministic in the untyped calculus $a$ must be $[V_i]$. Hence, we may take $V_i$ for $M'$.
Proof of Lemma 21

Assume $\Gamma \vdash M : \tau$ and $M \rightarrow M'$. We reason by induction on the proof of reduction.

Case $(\Lambda(\alpha) M_0) \tau \rightarrow_i M_0[\tau/\alpha]$: 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 $\proj_i \tau_1 \tau_2 (V_1, V_2) \rightarrow_\delta V_i$: 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 $z$ is $i$ or $\rightarrow_z$ otherwise) holds between $[M]$ and $[N]$. By rule congruence for $i$ and rule Context otherwise, the same relation holds between $[E[M]]$ and $[E][[N]]$, i.e. between $[M]$ and $[N]$.

Proof of Lemma 25

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 26

By Lemma 24, $M$ 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 25, $M_0$ a value $V_0$. Since $V_0 []$ is an evaluation context, $M_2$ is in $\iota$-normal form.

By 25 it must be a value $V_0$.

Moreover, by Lemma 25 $V_0$ is either

- $\Lambda\overline{\tau} \lambda x: \tau. M_1$. Since $V_0$ is in application position $\overline{\tau}$ must actually be empty. Then $M$ is of the form $R[\Lambda\overline{\alpha}.(\lambda x: \tau. M_1) V]$, as expected.
\[ R_0[c \bar{\tau} V_1 \ldots V_{n-1}] \]. Since \( V_0 \) is in an evaluation, \( R_0 \) is \( \iota \)-normal, thus of the form \( \Lambda \bar{\alpha}.[] \bar{\tau}_0 \). Since \( V_0 \) in application position \( \bar{\alpha} \) must be empty. From the arity of \( d \), the application is partial and has an arrow type, hence \( \bar{\tau}_0 \) must be empty. Then, taking \( V \) for \( V_n \), the term \( M \) is of the form \( R[c \bar{\tau} V_1 \ldots V_n] \), as expected.

Solution of Exercise 34

Take \((\lambda x: \tau_0. \Lambda \alpha. \lambda \alpha : y. y) M_0 \) for \( M \) where \( \Gamma \vdash M_0 : \tau_0 \). We have \( \Gamma \vdash M : \forall \alpha. \alpha \to \alpha \) where \( M \) is syntactically in ML, but cannot be typed in ML because function bodies cannot have polymorphic types.
Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.


François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL’11)*, Austin, Texas, January 2011. [Supplementary material](#).


