Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of December 18, 2014
## 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 λ-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

8.2 Mini Haskell .................................................. 153
  8.2.1 Examples in MH ........................................... 153
  8.2.2 The definition of Mini Haskell ......................... 154
  8.2.3 Semantics of Mini Haskell ............................... 156
  8.2.4 Elaboration of expressions ............................ 158
  8.2.5 Summary of the elaboration.......................... 159
  8.2.6 Elaboration of dictionaries ............................ 161

8.3 Implicitly-typed terms .................................... 163

8.4 Variations .................................................... 169
Chapter 1

Introduction

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

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

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

1.1 Overview of the course

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

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

Therefore, the course starts with a few reminders about the untyped λ-calculus, even though those are assumed to be known. We show how to extend the pure λ-calculus with constants and primitives and a few other constructs to make it a small programming language. This is also an opportunity to present source program transformations and compi-

---

1 Master Parisian de Recherche en Informatique.
lation techniques for function languages, which do not depend much on types. This part is taught by Xavier Leroy.

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

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

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

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

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

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

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

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

1.3 About Functional Programming

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

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

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

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

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

1.4 About Types

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

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

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

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

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

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

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

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

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

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

1.5 Acknowledgment

These course notes are based on and still contain a lot of material from a previous course taught for several years by François Pottier.
Chapter 2

The untyped $\lambda$-calculus

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

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

2.1 Syntax

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

$$a :: x | \lambda x. a | a a | \ldots$$

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

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

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

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

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

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

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

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

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

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

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

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

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

\(^1\)without lost of generality.
For example, we may define
\[
\text{let } h = PApp (PFun (\text{fun } f \rightarrow PApp (PVar f, PVar f)), PFun (\text{fun } x \rightarrow PVar x))
\]

Notice that \( h \) is polymorphic in the type of term variables. What term of the \( \lambda \)-calculus does it represent?  
(Solution p. 18)

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

## 2.2 Semantics

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

### 2.2.1 Strong v.s. weak reduction strategies

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

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

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

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

However, the metatheoretical properties, such as type soundness, are often simpler to establish for weak reductions strategies. Despite some advantages of the two step-approach to the semantics of programming languages, we will not pursue it here. We instead directly start with a weak reduction strategy. Still, we will informally discuss at certain places some of the properties that would hold if we had followed the more general approach.

## 2.2.2 Call-by-value semantics

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

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

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

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

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

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

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

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

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

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

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

Recursion

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

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

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

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

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

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

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

(Solution p. 19)
2.3 Answers to exercises

Solution of Exercise 1

type var = string
type term =
  | Var of var
  | Fun of var * term
  | App of term * term

Define in this abstract syntax the term funaa

Solution of Exercise 2

\((\lambda f. f f)(\lambda x. x)\).

Solution of Exercise 2, Question 2

\[\text{let gensym = let } n = \text{ref } 0 \text{ in fun () } \rightarrow \text{incr } n; \text{"x" } \rightarrow \text{string_of_int }!n;\]
\[\text{let rec to_term = function}\]
\[\text{  | PFun } f \rightarrow \text{let } x = \text{gensym()} \text{ in Fun } (x, \text{to_term } (f \text{ Var } x))\]
\[\text{  | PApp } (f, g) \rightarrow \text{App } (\text{to_term } f, \text{to_term } g)\]
\[\text{  | PVar } x \rightarrow x\]
\[\text{let } t = \text{to_term } h\]

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

Solution of Exercise 3

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

\[e ::= [\ ] a\]

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

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

Solution of Exercise 4

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

- **Eval-Fun**
  
  $E \vdash \lambda x. a \leadsto \lambda x. a$

- **Eval-Var**
  
  $x \mapsto v \in E$
  
  $E \vdash x \leadsto v$

- **Eval-App**
  
  $E \vdash a_1 \leadsto \lambda x. a$
  
  $E \vdash a_2 \leadsto v$
  
  $x \mapsto v \in E$
  
  $E, x \vdash a \leadsto v$
  
  $E \vdash a_1 a_2 \leadsto v$

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

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

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

Solution of Exercise 7

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

```ocaml
let zfix g = (fun x -> x x) (fun z -> g (fun v -> z z v))
let gfact f n = if n > 0 then n * f (n-1) else 1
let fact = zfix gfact;;
let six = fact 3;;
```

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

Simply-typed lambda-calculus

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

3.1 Syntax

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

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

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

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

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

3.2 Dynamic semantics

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

$$V ::= \lambda x : \tau. M | \ldots$$

$$E ::= \emptyset M | V \emptyset | \ldots$$
The reduction relation $M_1 \longrightarrow M_2$ is inductively defined by:

\[
\begin{align*}
\beta_v & (\lambda x: \tau. M) V \longrightarrow [x \mapsto V]M \\
\text{Context} & \\
M \longrightarrow M' & \longrightarrow E[M] \longrightarrow E[M']
\end{align*}
\]

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

### 3.3 Type system

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

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

Typing judgments are defined inductively by the following inference rules:

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash x : \Gamma(x) \\
\text{Abs} & \quad \Gamma, x : \tau_1 \vdash M : \tau_2 \\
& \quad \Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2 \\
\text{App} & \quad \Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \\
& \quad \Gamma \vdash M_2 : \tau_2 \\
& \quad \Gamma \vdash M_1 \ M_2 : \tau_2
\end{align*}
\]

By our convention on well-formedness of typing contexts, the premise of rule [ABS] carries the implicit assumption $x \notin \text{dom}(\Gamma)$. This condition can always be satisfied, since $x$ is bound in the expression $\lambda x : \tau. M$ and can be renamed if necessary.

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

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

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

```
VAR
\[ \Gamma \vdash f : \tau_1 \rightarrow \tau_2 \]

VAR
\[ \Gamma \vdash g : \tau_0 \rightarrow \tau_1 \quad \Gamma \vdash x : \tau_0 \]

APP
\[ \Gamma \vdash g \, x : \tau_1 \]

VAR
\[ \Gamma \vdash f \, (g \, x) : \tau_2 \]

APP
\[ \Gamma \vdash f \, (g \, x) : \tau_2 \]

ABS
\[ f : \tau_1 \rightarrow \tau_2, g : \tau_0 \rightarrow \tau_1 \vdash \lambda x : \tau_0 \cdot f \, (g \, x) : \tau_0 \rightarrow \tau_2 \]

ABS
\[ \emptyset \vdash \lambda f : \tau_1 \rightarrow \tau_2, \lambda g : \tau_0 \rightarrow \tau_1, \lambda x : \tau_0 \cdot f \, (g \, x) : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_0 \rightarrow \tau_1) \rightarrow \tau_0 \rightarrow \tau_2 \]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\lceil x \rceil & \triangleq x \\
\lceil \lambda x: \tau. M \rceil & \triangleq \lambda x. \lceil M \rceil \\
\lceil M_1 \ M_2 \rceil & \triangleq \lceil M_1 \rceil \lceil M_2 \rceil
\end{align*}
\]

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

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

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

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

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

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

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

\[ M_1 \xrightarrow{\beta} M_2 \]
\[ a_1 \xrightarrow{\beta} a_2 \]
\[ \| \]
\[ \| \]

Direct simulation

\[ M_1 \rightarrow M_2 \]
\[ a_1 \rightarrow a_2 \]
\[ \| \]
\[ \| \]

Inverse simulation

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

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

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

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

(Solution p. 44)

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

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

### 3.4 Type soundness

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

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

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

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

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

\subsection{Proof of subject reduction}

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

\textbf{Type preservation by \( \beta \)-reduction.}

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

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

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

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

\[
\frac{
\frac{\text{ABS}}{x : \tau \vdash M : \tau_0 \quad (4)}}{
\frac{\text{APP}}{\emptyset \vdash (\lambda x : \tau. M) : \tau \rightarrow \tau_0 \quad (5)}}{
\emptyset \vdash (\lambda x : \tau. M) \ V : \tau_0 \quad (2)}
\]

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

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

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

\textbf{Lemma 5 (Value substitution, strengthened)} If \( x : \tau, \Gamma \vdash M : \tau_0 \) and \( \emptyset \vdash V : \tau \), then \( \Gamma \vdash [x \mapsto V] M : \tau_0 \).
The proof is then straightforward provided we have a *weakening* lemma (stated below) in the case for variables. (In the case for abstraction, the variable for the parameter can—and must—be chosen different from the variable $x$.) This closes the $\beta$-reduction proof case for type preservation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

**3.4. TYPE SOUNDNESS**

27
Type preservation by reduction under an evaluation context.

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

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

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

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

The proof is by cases over $E$; each case is straightforward.

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

3.4.2 Proof of progress

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

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

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

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

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

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

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

- ...

Proof: By cases over $V$:

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

- ...

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

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

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

This calls for mechanized proofs that ensure case coverage while trivial cases should be automatically dischargeable.

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

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

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

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

(Solution p. 44)

3.5 Simple extensions

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

3.5.1 Unit

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

$$\tau ::= \ldots \mid \text{unit} \quad V ::= \ldots \mid () \quad M ::= \ldots \mid ()$$

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

$$\text{UNIT} \quad \frac{}{\Gamma \vdash () : \text{unit}}$$

Exercise 13 Check that type soundness is preserved. 

(Solution p. 44)

Notice that the classification Lemma is no longer degenerate.

3.5.2 Boolean

$$V ::= \ldots \mid \text{true} \mid \text{false} \quad M ::= \ldots \mid \text{true} \mid \text{false} \mid \text{if } M \text{ then } M \text{ else } M$$

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

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

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

$$\begin{align*}
\text{if true then } M_1 \text{ else } M_2 & \rightarrow M_1 \\
\text{if false then } M_1 \text{ else } M_2 & \rightarrow M_2
\end{align*}$$

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

$$\tau ::= \ldots \mid \text{bool}$$
3.5. SIMPLE EXTENSIONS

The new typing rules are:

\[
\begin{align*}
\text{True} & : \epsilon \\
\Gamma & \vdash \text{true} : \text{bool} \\
\text{False} & : \epsilon \\
\Gamma & \vdash \text{false} : \text{bool} \\
\text{IfThenElse} & : \epsilon \\
\Gamma & \vdash \text{if } M_0 \text{ then } M_1 \text{ else } M_2 : \tau \\
\end{align*}
\]

Exercise 14 Given the new cases for the classification lemma (without proving them). Check that progress is preserved.

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

3.5.3 Pairs

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

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

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

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

\[
\text{proj}_i (V_1, V_2) \longrightarrow V_i
\]

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

\[
\begin{align*}
\tau & ::= \ldots | \tau \times \tau \\
\text{PAIR} & : \Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2 \\
\Gamma & \vdash (M_1, M_2) : \tau_1 \times \tau_2 \\
\text{PROJ} & : \Gamma \vdash M : \tau_1 \times \tau_2 \\
\Gamma & \vdash \text{proj}_i M : \tau_i
\end{align*}
\]

Exercise 16 Check that subject reduction is preserved when adding pairs.

Exercise 17 Modify the semantics to evaluate pairs from right to left. Would this be sound? Would this be still call-by-value?
3.5.4 Sums

Values, expressions, evaluation contexts are extended:

\[
M ::= \ldots | \text{inj}_i M | \text{case } M \text{ of } V \circ V
\]

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

\[
E ::= \ldots | \text{inj}_i [] | \text{case } [] \text{ of } V \circ V
\]

A new reduction rule is introduced:

\[
\text{case } \text{inj}_i V \text{ of } V_1 \circ V_2 \rightarrow V_i V
\]

Sum types are added to classify sums:

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

Two new typing rules are introduced:

\[
\dfrac{
\Gamma \vdash M : \tau_i
}{
\Gamma \vdash \text{inj}_i M : \tau_1 + \tau_2}
\]

\[
\dfrac{
\begin{align*}
\Gamma \vdash M : \tau_1 + \tau_2 \\
\Gamma \vdash V_1 : \tau_1 \rightarrow \tau \\
\Gamma \vdash V_2 : \tau_2 \rightarrow \tau
\end{align*}
}{
\Gamma \vdash \text{case } M \text{ of } V_1 \circ V_2 : \tau}
\]

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

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

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

Exercise 18 Describe the extension with the option type.

3.5.5 Modularity of extensions

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

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

Then, in each case,
3.5. SIMPLE EXTENSIONS

- subject reduction is preserved because types of new redexes are preserved by the new reduction rules.
- progress is preserved because the type system ensures that the new destructors can only be applied to values such that at least one of the new reduction rules applies.

Moreover, the extensions are independent: they can be added to the \( \lambda \)-calculus alone or mixed altogether. Indeed, no assumption about other extensions (the “...”) has ever been made, except for the classification lemma which requires, informally, that \textit{values of other shapes have types of other shapes}. This is obviously the case in the extensions we have presented: the unit has the unit type, pairs have product types, and sums have sum types.

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

3.5.6 Recursive functions

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

\[
V ::= \ldots | \mu f:\tau.\lambda x.M \\
M ::= \ldots | \mu f:\tau.\lambda x.M
\]

A new reduction rule is introduced to unfold recursive calls:

\[
(\mu f:\tau.\lambda x.M) V \rightarrow [f \mapsto \mu f:\tau.\lambda x.M][x \mapsto V]M
\]

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

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

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

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

3.5.7 A derived construct: let-bindings

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

\[
\begin{align*}
\text{ABS} & \quad \frac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1. M_2 : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash M_1 : \tau_1}{\Gamma \vdash (\lambda x : \tau_1. M_2) M_1 : \tau_2}
\end{align*}
\]

This means that the following derived rule is sound and complete for let-bindings:

\[
\begin{align*}
\text{LETMONO} & \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = M_1 \text{ in } M_2 : \tau_2}
\end{align*}
\]

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

\[
\begin{align*}
\text{LETMONO} & \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}
\end{align*}
\]

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

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

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

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

\[
\begin{align*}
\text{FIXABS} & \quad \frac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \mu f : \tau \to \tau_1. \lambda x. M_1 : \tau \to \tau_1} \quad \frac{\Gamma \vdash f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let rec } (f : \tau \to \tau_1) \ x = M_1 \text{ in } M_2 : \tau_2}
\end{align*}
\]

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

\[
\begin{align*}
\text{LETRECMONO} & \quad \frac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1 \quad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \text{let rec } (f : \tau \to \tau_1) \ x = M_1 \text{ in } M_2 : \tau_2}
\end{align*}
\]
3.6 Exceptions

Exceptions are a mechanism for changing the normal order of evaluation (usually, but not necessarily, in case something abnormal occurred).

When an exception is raised, the evaluation does not continue as usual: Shortcutting normal evaluation rules, the exception is propagated up into the evaluation context until some handler is found at which the evaluation resumes with the exceptional value received; if no handler is found, the exception reaches the toplevel and the result of the evaluation is the exception instead of a value.

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

3.6.1 Semantics

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

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

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

\begin{align*}
\text{Raise} & : F[\text{raise } V] \rightarrow \text{raise } V \\
\text{Handle-Val} & : \text{try } V \text{ with } M \rightarrow V \\
\text{Handle-Raise} & : \text{try raise } V \text{ with } M \rightarrow M V
\end{align*}

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

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

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

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

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

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

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

\[
\text{RAISE} \quad \frac{\Gamma \vdash M : \text{exn}}{\Gamma \vdash \text{raise } M : \tau}
\]

\[
\text{TRY} \quad \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \text{exn} \rightarrow \tau}{\Gamma \vdash \text{try } M_1 \text{ with } M_2 : \tau}
\]

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

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

This is the approach followed by ML. However, since the set of exceptions is not known in advance, ML declares a new type $\text{exn}$ for exceptions and allows adding new cases to the sum later on as needed. This is called an extensible datatype. (Until recently, the type of exceptions was the only extensible datatypes in OCaml, but since version 4.02, the user may define his own.)

As a counterpart checking for exceptions can’t be exhaustive without a “catch all” branch, since further cases could always be added later. Notice that although new constructors may be added, the type of exception is fixed in the whole program, to $\text{exn}$. This is essential for type soundness, since the handling and raising of exceptions must agree globally on the type $\text{exn}$ of exceptional values as it is not passed around.

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

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

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

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

(Solution p. 40)
3.6. EXCEPTIONS

3.6.3 Variations

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

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

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

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

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

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

\begin{align*}
\text{Handle-Val'}: & \quad \text{try } V \text{ with } M \rightarrow V \\
\text{Handle-Raise'}: & \quad \text{try } F[\text{raise } V] \text{ with } M \rightarrow M V
\end{align*}

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

Exceptions with bindings Benton and Kennedy (2001) have argued for merging let-bindings with exception handling into a unique form \texttt{let } \( x = M_1 \text{ with } M_2 \) in \( M_3 \). The expression \( M_1 \) is evaluated first and, if it returns a value, it is substituted for \( x \) in \( M_3 \), as if we had evaluated \texttt{let } \( x = M_1 \) in \( M_3 \); otherwise, i.e., if it raises an exception \texttt{raise } \( V \), then the exception is handled by \( M_2 \), as if we had evaluated \texttt{try } \( M_1 \) with \( M_2 \).
This combined form captures a common pattern in programming that has no elegant workaround:

\[
\text{let rec read\_config\_in\_path filename (dir :: dirs) →}
\\	ext{let fd = open\_in (Filename.concat dir filename)}
\\	ext{with Sys\_error _ → read\_config filename dirs in}
\\	ext{read\_config\_from\_fd fd}
\]

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

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

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

\[
\text{case (try Val } M_1 \text{ with } \lambda y. \text{Exc } y \text{) of (Val: } \lambda x. M_3 \circ \text{Exc: } M_2)
\]

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

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

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

**Exercise 21** Describes the dynamic semantics of the \( \text{let } x = M_1 \text{ with } M_2 \text{ in } M_3 \) construct, formally. (Solution p. 46)

**Exercise 22 (try finalize)** A finalizer is some code that should be run in case of both normal and exceptional evaluation. Write a function finalize that takes four arguments \( f \), \( x \), \( g \), and \( y \) and returns the application \( f \ x \) with finalizing code \( g \ y \). i.e. \( g \ y \) should be called before returning the result of the application of \( f \) to \( x \) whether it executes normally or raises an exception. (Solution p. 46)

A similar construct has been added in OCaml, version 4.02, allowing exceptions to be combined with pattern matching. The previous example can now be written:

\[
\text{let rec read\_config\_in\_path filename (dir :: dirs) →}
\\	ext{match open\_in (Filename.concat dir filename) with}
\\	ext{| fd → read\_config\_from\_fd fd}
\\	ext{| exception Sys\_error _ → read\_config filename dirs}
\]
3.7 References

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

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

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

\[ (\lambda x: \tau. (x := 1; ! x)) \text{(ref 3)} \]

which intuitively should yield 1, would reduce to:

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

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

3.7.1 Language definition

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

\[ M ::= \ldots | \ell \]
\[ V ::= f \ldots | \ell \]

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

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

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

\[ (\lambda x: \tau. M) V / \mu \rightarrow [x \mapsto V] M / \mu \]
\[ E[M] / \mu \rightarrow E[M'] / \mu' \quad \text{if } M / \mu \rightarrow M' / \mu' \]
Three new reduction rules are added:

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

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

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

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

Three new typing rules are introduced:

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

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

Intuitively, this could be formalized by a typing rule of the form:

\[
\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \text{ref } \tau}
\]

Then, typing judgments would have the form \( \mu, \Gamma \vdash M : \tau \). Typing judgments would no longer be \textit{inductively} defined (or else, every cyclic structure would be ill-typed). Instead, \textit{co-induction} would be required. Moreover, if the value \( \mu(\ell) \) happens to admit two distinct types \( \tau_1 \) and \( \tau_2 \), then \( \ell \) admits types \text{ref } \tau_1 and \text{ref } \tau_2. \text{So, one can write at type } \tau_1 \text{ and read at type } \tau_2: \text{ this rule is \textit{unsound}!}

A simpler, and sound, approach is to fix the type of a memory location when it is first allocated. To do so, we use a \textit{store typing} \( \Sigma \), a finite mapping of locations to types. Then, a location \( \ell \) has type \text{ref } \tau \text{ \textit{when the store typing } } \Sigma \text{ \textit{says so.}}

\[
\frac{\Sigma, \varnothing \vdash \ell : \text{ref } \Sigma(\ell)}{\Sigma \vdash \ell : \text{ref } \Sigma(\ell)}
\]

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

\[
\begin{align*}
\text{Store} \quad & \forall \ell \in \text{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell) \\
\text{Config} \quad & \Sigma, \varnothing \vdash M : \tau \quad \vdash \mu : \Sigma
\end{align*}
\]

Remarks:

\[\text{This could happen, for example, in the presence of sum types (described in \S 3.5.4), when expressions do not have unique types any longer.}\]
3.7. REFERENCES

- This is an inductive definition. The store typing Σ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.

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

3.7.2 Type soundness

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

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

**Theorem 6 (Progress)** If $M \downarrow \mu$ is a well-typed, irreducible configuration, then $M$ is a value.

Inlining subject reduction can also be restated as:

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

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

$M \downarrow \mu \rightarrow M' \downarrow \mu'$ and $\Sigma, \emptyset \vdash E[M] : \tau$ and $\vdash \mu : \Sigma$

Assuming compositionality, there exists $\tau'$ such that:

$\Sigma, \emptyset \vdash M : \tau'$ and $M'$, $(\Sigma, \emptyset \vdash M' : \tau') \Rightarrow (\Sigma, \emptyset \vdash E[M'] : \tau)$

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

$\Sigma', \emptyset \vdash M' : \tau'$ and $\vdash \mu' : \Sigma'$

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

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

At each reduction step, the new store typing $\Sigma'$ extends the previous store typing $\Sigma$. Growing the store typing preserves well-typedness (a generalization of the weakening lemma):
Lemma 11 (Stability under memory allocation) If $\Sigma \subseteq \Sigma'$ and $\Sigma, \Gamma \vdash M : \tau$, then $\Sigma', \Gamma \vdash M : \tau$.

This allows establishing a strengthened version of compositionality:

Lemma 12 (Compositionality) Assume $\Sigma, \emptyset \vdash E[M] : \tau$. Then, there exists $\tau'$ such that:

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

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

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

By compositionality, there exists $\tau'$ such that:

$$\Sigma, \emptyset \vdash M : \tau'$$

$$\forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \emptyset \vdash M' : \tau') \Rightarrow (\Sigma', \emptyset \vdash E[M'] : \tau')$$

By the induction hypothesis, there exists $\Sigma'$ such that:

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

The goal immediately follows.

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

3.7.3 Tracing effects with a monad

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

```
return : \alpha \to \text{IO} \alpha
bind : \text{IO} \alpha \to (\alpha \to \text{IO} \beta) \to \text{IO} \beta
main : \text{IO} ()
newIORRef : \alpha \to \text{IO} (\text{IORRef} \alpha)
readIORRef : \text{IORRef} \alpha \to \text{IO} \alpha
writeIORRef : \text{IORRef} \alpha \to \alpha \to \text{IO} ()
```

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

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

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

Further reading

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

Solution of Exercise 8

See the statement of bisimilation for System-F in §4.4.5 in particular lemmas 21 and 33.

Solution of Exercise 10

Case $M$ is $M_1 \ M_2$: By inversion of the judgment $\Gamma \vdash M : \tau$, we must have $\Gamma \vdash M_1 : \tau_2 \to \tau$ and $\Gamma \vdash M_2 : \tau_2$ for some $\tau_2$. By induction hypothesis, we have $\Gamma, y : \tau' \vdash M_1 : \tau_2 \to \tau$ and $\Gamma, y : \tau' \vdash M_2 : \tau_2$, respectively. We conclude by an application of Rule APP.

Solution of Exercise 11

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

Solution of Exercise 12

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

Subject reduction can then be stated as $(\longrightarrow) \subseteq (\subset)$.

Proof: Since $(\longrightarrow)$ is the smallest relation that satisfies rules BETA and CONTEXT it suffices to show that $\subseteq$ also satisfies rules BETA and CONTEXT.

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

Case CONTEXT: Assume $M \subseteq M'$. Let us show $E[M] \subseteq E[M']$. Assume $\Gamma \vdash E[M] : \tau$. Then $\Gamma \vdash E[M'] : \tau$ follows by compositinality.

Solution of Exercise 13

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

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

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

**Solution of Exercise 14**

The new case for the classification Lemma is that a value of type `bool` must be a boolean, i.e. either `true` or `false` (5).

For the proof of progress, we assume that `∅ ⊢ M : τ` (6) and show that `M` is either a value or reducible (4??) by structural induction on `M`. We have two new cases:

*Case `M` is `true` or `false`: In both cases, `M` is a value.*

*Case `M` is `if M_0 then M_1 else M_2`: By inversion of typing rules applied to (6), we have `∅ ⊢ M_0 : bool`, `∅ ⊢ M_1 : τ`, and `∅ ⊢ M_2 : τ`. If `M_0` is a value, then, since it is of type `bool`, it must be `true` or `false` by (5), and in both cases, `M` reduces by either one of the two new rules. Otherwise, by induction hypothesis, `M_0` must be reducible, and so is `M` by rule CONTEXT since if `[]` then `M_1` else `M_2` is an evaluation context. This ends the proof.

**Solution of Exercise 15**

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

\[
V ::= \ldots | \bar{n} \\
M ::= \ldots | n | M + M | M \times M
\]

We add only evaluation contexts:

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

two reduction rules are:

\[
\bar{n} + \bar{m} \longrightarrow \bar{n + m} \\
\bar{n} \times \bar{m} \longrightarrow \bar{n \times m}
\]

and the following typing rules:

\[
\text{INT} \\
\Gamma \vdash \bar{n} : \text{int} \\
\text{PLUS} \\
\Gamma \vdash M_1 : \text{int} \quad \Gamma \vdash M_2 : \text{int} \\
\Gamma \vdash M_1 + M_2 : \text{int} \\
\text{TIMES} \\
\Gamma \vdash M_1 : \text{int} \quad \Gamma \vdash M_2 : \text{int} \\
\Gamma \vdash M_1 \times M_2 : \text{int}
\]

**Solution of Exercise 16**

The proof of subject reduction is by cases on the reduction rule. We have two new reduction rules for each the projection, which can be factorized as follows:

\[
\text{proj}_i (V_1, V_2) \longrightarrow
\]
We assume that $\Gamma \vdash \text{proj}_i (V_1, V_2) : \tau$ (2??). By inversion of typing of judgment, we know that the derivation of (2) ends with:

$$\begin{array}{c}
\text{PAIR} \\
\frac{}{\Gamma \vdash V_1 : \tau_1 \quad \Gamma \vdash V_2 : \tau_2} \quad \frac{}{\Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2}
\end{array}$$

with $\tau$ of the form $\tau_1 \rightarrow \tau_2$. We must show that $\Gamma \vdash V : i \tau_i$ which is either one of the hypotheses (1) or (3).

**Solution of Exercise 17**

Just exchange $M$ and $V$ in the definition of evaluation contexts. This does not break soundness of course. The semantics is still call-by-value.

**Solution of Exercise 20**

No, because exceptions allow to hide the type of values that they communicate, and one may create a recursion without noticing it from types.

For instance, take the type $\text{exn} \equiv \tau \rightarrow \tau$ where $\tau$ is $\text{unit} \rightarrow \text{unit}$. You may then define the inverse coercion functions between types $\tau \rightarrow \tau$ and $\tau$:

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

Therefore, we may define the term $\omega$ as $\lambda x. (\text{unfold } x) \ x$ and the term $\omega$ ($\text{fold } \omega$) whose reduction does not terminate.

**Solution of Exercise 21**

We need a new evaluation context:

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

and the following reduction rules:

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

**Solution of Exercise 22**
let finalize f x g y =
  let result = try f x with exn -> g y; raise exn in
  g y; result

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

type 'a result = Val of 'a | Exc of exn
let finalize f x g y =
  let result = try Val (f x) with exn -> Exc exn in
  g y;
  match result with Val x -> x | Exc exn -> raise exn

As a counterpart, this allocated an intermediate result.
Bibliography

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


