Type systems for programming languages

Didier Rémy

Academic year 2013-1024
Version of November 11, 2013

Contents

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[1]) 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 λ-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 $\mathsf{HM(X)}$ 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, λ-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 λ-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; $\mathsf{int} \to \mathsf{bool}$ describes a function that maps an integer argument to a boolean result; $(\mathsf{int} \to \mathsf{bool}) \to (\mathsf{list}\ \mathsf{int} \to \mathsf{list}\ \mathsf{int})$ 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 sophiscated types (*e.g.* with dependent types, logical assertions, afine types, capabililties 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 λ-calculus

In this course, λ-calculus is the underlying computational language. The λ-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 λ-calculus, most of our results can also be applied to other languages (Java, assembly language, *etc.*).

The untyped λ-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 λ-*terms,* also known as *terms* and *expressions,* are given by the grammar:

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

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

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

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

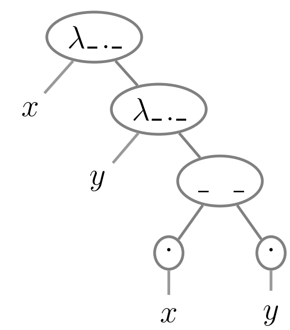
$$a ::= x \mid \lambda x.\, a \mid a\; a \mid \mathsf{let}\ x = a\ \mathsf{in}\ a$$

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 λ-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 variatons 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 λ-calculus.*

(Solution p. 18) □

Exercise 2 *Higher Order Abstract Syntax (HOAS) uses the binding and α-conversion mechanisms of the host language (here* OCaml*) to implement bindings and α-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

$$\texttt{let}\ h = PApp\ (PFun\ (\texttt{fun}\ f \to PApp\ (PVar\ f,\ PVar\ f)),\ PFun\ (\texttt{fun}\ x \to PVar\ x))$$

Notice that h *is polymorephic in the type of term variables. What term of the* λ*-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 λ-calculus is given by a *small-step operational* semantics, *i.e.* a reduction relation between λ-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 λ-calculus, one can choose a *strong* notion of reduction, *i.e.* that can perform reduction in any context, in particular under λ-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 λ-calculus is confluent: for terms that are strongly normalizing, *i.e.* do not contain infinite reduction path, then all possible reduction paths end up on the same normal form: the calculus is confluent.

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

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

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

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

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

2.2.2 Call-by-value semantics

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

In the pure λ-calculus, the *values* are the functions:

$$v ::= \lambda x.\, a \mid \dots$$

Variables are not values in the call-by0value λ-calculus. We only evaluate closed terms, hence a variable should never appear in an evaluation context. Notice that any function is a value in the *call-by-value* λ-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 β-redex.

The *reduction relation* $a_1 \longrightarrow a_2$ is inductively defined:

$$\beta_v \qquad (\lambda x.\, a)\ v \longrightarrow [x \mapsto v]a \qquad\qquad \textsc{Context} \qquad \frac{a \longrightarrow a'}{e[a] \longrightarrow e[a']}$$

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

$$e ::= [\,]\ a \mid v\ [\,] \mid \dots$$

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

$$\bar{e} ::= [\,] \mid e[\bar{e}]$$

Exercise 3 *Define the semantics of the call-by-name λ-calculus.* (Solution p. 18) □

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

Exercise 5 *Write an interpreter for a call-by-value λ-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 λ-calculus, hence reduction may not terminate. For example, the term $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ known as Δ reduces to itself, and so may reduce forever.

A slight variation on Δ 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 η-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

```
let gensym = let n = ref 0 in fun () → incr n; "x" ^ string_of_int !n;;
let rec to_term = function
  | PFun f → let x = gensym() in Fun (x, to_term (f (Var x)))
  | PApp (f, g)  → App (to_term f, to_term g)
  | PVar x → x
let t = to_term h
```

```
val t : term = App (Fun ("x2", App (Var "x2", Var "x2")), Fun ("x1", Var "x1"))
```

■

Solution of Exercise 3

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

$$e ::= [\,]\ a$$

As a counterpart, β-reduction must not require its argument to be evaluated. Hence the call-by-name β_n rule is:

$$(\lambda x.\, a_0)\ a \longrightarrow [x \mapsto a]a_0 \qquad (\beta_n)$$

■

Solution of Exercise 4

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

$$\text{EVAL-FUN} \quad \mathcal{E} \vdash \lambda x.\, a \rightsquigarrow \lambda x.\, a \qquad \text{EVAL-VAR} \quad \frac{x \mapsto v \in \mathcal{E}}{\mathcal{E} \vdash x \rightsquigarrow v} \qquad \text{EVAL-APP} \quad \frac{\mathcal{E} \vdash a_1 \rightsquigarrow \lambda x.\, a \qquad \mathcal{E} \vdash a_2 \rightsquigarrow v \qquad \mathcal{E}, x \mapsto v \vdash a \rightsquigarrow v}{\mathcal{E} \vdash a_1\, a_2 \rightsquigarrow v}$$

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 λ-bound variable `fun` $x \to x\ x$. In OCaml, this is ill-typed, as it requires x to have both types α and $\alpha \to \beta$ simultaneously, which is only possible if α is a recursive type $(\ldots(\alpha \to \ldots) \to \alpha)$ With the `-rectype` option, one can defined:

```
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.

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

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

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

where α denotes a type variable. We assume given a denumerable collection of type variables. This definition says that a type τ is a type variable α, 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:

$$V ::= \lambda x\!:\!\tau.\, M \mid \ldots \qquad\qquad E ::= [\,]\ M \mid V\ [\,] \mid \ldots$$

The *reduction relation* $M_1 \longrightarrow M_2$ is inductively defined by:

$$\overset{\beta_v}{(\lambda x{:}\tau.\, M)\; V \longrightarrow [x \mapsto V]M} \qquad\qquad \overset{\text{Context}}{\dfrac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}}$$

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* Γ (also called a typing environment) is a finite sequence of bindings of program variables to types. The empty context is written $\varnothing$. A typing context Γ can be extended with a new binding τ for x with the notation $\Gamma, x : \tau$. To avoid confusion between the new binding and any other binding that may appear in Γ, 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 $\mathsf{dom}(\Gamma)$ for the set of variables bound by Γ and $\Gamma(x)$ for the type τ bound to x in Γ, which implies that x is in $\mathsf{dom}(\Gamma)$. We write $x : \tau \in \Gamma$ to mean that Γ maps x to τ, and $x \# \mathsf{dom}(\Gamma)$ to mean that $x \notin \mathsf{dom}(\Gamma)$.

Typing judgments are defined inductively by the following inference rules:

$$\overset{\text{Var}}{\Gamma \vdash x : \Gamma(x)} \qquad \overset{\text{Abs}}{\dfrac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2}} \qquad \overset{\text{App}}{\dfrac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2}}$$

By our convention on well-formedness of typing contexts, the premise of rule Abs carries the implicit assumption $x \# \mathsf{dom}(\Gamma)$. 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 Γ stands for $f : \tau_1 \to \tau_2; g : \tau_0 \to \tau_1; x : \tau_0$.

$$\textsc{Abs}\;\dfrac{\textsc{Abs}\;\dfrac{\textsc{Abs}\;\dfrac{\textsc{App}\;\dfrac{\overset{\textsc{Var}}{\Gamma \vdash f : \tau_1 \to \tau_2} \qquad \dfrac{\overset{\textsc{Var}}{\Gamma \vdash g : \tau_0 \to \tau_1} \qquad \overset{\textsc{Var}}{\Gamma \vdash x : \tau_0}}{\Gamma \vdash g\ x : \tau_1}\;\textsc{App}}{\Gamma \vdash f\ (g\ x) : \tau_2}}{f : \tau_1 \to \tau_2, g : \tau_0 \to \tau_1 \vdash \lambda x{:}\tau_0.\, f\ (g\ x) : \tau_2}}{f : \tau_1 \to \tau_2 \vdash \lambda g{:}\tau_0 \to \tau_1.\, \lambda x{:}\tau_0.\, f\ (g\ x) : \tau_2}}{\varnothing \vdash \lambda f{:}\tau_1 \to \tau_2.\, \lambda g{:}\tau_0 \to \tau_1.\, \lambda x{:}\tau_0.\, f\ (g\ x) : (\tau_1 \to \tau_2) \to (\tau_0 \to \tau_1) \to \tau_0 \to \tau_2}$$

This derivation is valid for any choice of τ_1 and τ_2. Conversely, every derivation for this term must have this shape, for some τ_1 and τ_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* $x \in \mathsf{dom}(\Gamma)$ *and* $\Gamma(x) = \tau$.
- *If* M *is* $M_1\ M_2$ *then* $\Gamma \vdash M_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash M_2 : \tau_2$ *for some type* τ_2.
- *If* M *is* $\lambda x{:}\tau_0.\, M_1$, *then* τ *is of the form* $\tau_0 \to \tau_1$ *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:

$$\begin{array}{rcl} \lceil x \rceil & \triangleq & x \\ \lceil \lambda x{:}\tau.\, M \rceil & \triangleq & \lambda x.\lceil M \rceil \\ \lceil M_1\ M_2 \rceil & \triangleq & \lceil M_1 \rceil \lceil M_2 \rceil \end{array}$$

The erasure of a term M of System 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 (forward simulation), and conversely, a reduction step after type erasure can also be traced back in the typed language as a reduction step between associated source terms (backward simulation). Formally, this can be stated as follows:

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

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

Diagramatically, we have

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Forward simulation

$$\begin{tikzcd} M_1 \arrow[r, dashed, "\beta"] \arrow[d, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, "\beta"'] & a_2 \end{tikzcd}$$

Backward simulation

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

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

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

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

(Solution p. 45)

□

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

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

3.4 Type soundness

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

Theorem 1 (Type Soundness) *Well-typed expressions do not go wrong.*

The proof of type soundness is by combination of *Subject Reduction* (Lemma 2) and *Progress* (Lemma 3). This syntactic proof method is due to Wright and Felleisen (1994).

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

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

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

3.4.1 Proof of subject reduction

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

Type preservation by β-reduction.

In the proof of subject reduction for the β-reduction case, the hypotheses are

$$(\lambda x{:}\tau.\, M)\; V \longrightarrow [x \mapsto V]M \quad \mathbf{(1)} \qquad\qquad \varnothing \vdash (\lambda x{:}\tau.\, M)\; V : \tau_0 \quad \mathbf{(2)}$$

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

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

$$\textsc{App}\;\dfrac{\textsc{Abs}\;\dfrac{x : \tau \vdash M : \tau_0 \ \mathbf{(4)}}{\varnothing \vdash (\lambda x{:}\tau.\, M) : \tau \to \tau_0} \qquad \varnothing \vdash V : \tau \ \mathbf{(5)}}{\varnothing \vdash (\lambda x{:}\tau.\, M)\; V : \tau_0 \ (2)}$$

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

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

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

Lemma 5 (Value substitution, strengthened) *If $x : \tau, \Gamma \vdash M : \tau_0$ and $\varnothing \vdash V : \tau$, 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 β-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 β-reduction is:

Lemma 6 (Weakening) *If* $\varnothing \vdash V : \tau_1$ *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-typednessed 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* $y \notin \mathsf{dom}(\Gamma)$, *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 τ in Γ. Hence, it is also bound to τ in $\Gamma, y : \tau'$. We conclude by rule Var.

Case M is $\lambda x : \tau_2 . M_1$: *W.l.o.g*, we may choose $x \notin \mathsf{dom}(\Gamma)$ and $x \neq y$. We have $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$ with $\tau_2 \to \tau_1$ equal to τ. 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 Abs.

Case M is $M_1\ M_2$: easy.

Exercise 10 *Write the details of the application case for weakening.* *(Solution p. 45)* □

Exercise 11 *Try to prove the unstrengthened version and see where you get stuck.* *(Solution p. 45)* □

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

The result is obvious since a permutation of Γ 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 Γ with type variable declarations). Formally, the proof is by induction on M.

Type preservation by reduction under an evaluation context.

The first hypothesis is $M \longrightarrow M'$ (**1**) where, by induction hypothesis, this reduction preserves types (**2**). The second hypothesis is $\varnothing \vdash E[M] : \tau$ (**3**) where E is an *evaluation context*. The goal is $\varnothing \vdash E[M'] : \tau$ (**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 $\varnothing \vdash E[M] : \tau$, then, there exists τ' such that:*

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

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

Remark 2 Informally, τ' is the type of the hole in the context E, itself of type τ; we could write the pseudo judgment $\varnothing \vdash E[\tau'] : \tau$. (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 λ-calculus, there are just three cases: variable; λ-abstraction; and application. The case of variables is void, since a variable is never well-typed in the empty environment. The case of λ-abstractions is immediate, because a λ-abstraction is a value. In the only remaining case of an application, we show that M is always reducible.

Assume that $\varnothing \vdash M : \tau_1$ and M is an application $M_1\ M_2$. By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \to \tau_1$ and $\varnothing \vdash M_2 : \tau_2$. By induction hypothesis, M_1 is either reducible or a value V_1. If M_1 is reducible, so is M because $[]\ M_2$ is an evaluation context and we are done. Otherwise, by induction hypothesis, M_2 is either reducible or a value V_2. If M_2 is reducible, so is M because $V_1\ []$ is an evaluation context and we are done. Otherwise, because V_1 is a value of type $\tau_1 \to \tau_2$, it must be a λ-abstraction by classification of values (Lemma 10, below), so $V_1\ V_2$ is a β-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 $\varnothing \vdash V : \tau$. Then,*

- *if τ is an arrow type, then V is a λ-abstraction;*
- ...

Proof: By cases over V:

- if V is a λ-abstraction, then τ 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 λ-calculus, classification is trivial, because *every value is a λ-abstraction.* Progress holds even in the absence of the well-typedness hypothesis, *i.e.* in the untyped λ-calculus, because *no term is ever stuck!*

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

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

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

Soundness versus completeness Because the λ-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 λ-calculus, since progress holds in λ-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 (Γ, τ) such that $\Gamma \vdash M : \tau$. Define a relation $\sqsubseteq$ on typings such that $M \sqsubseteq M'$ means that all typings of M are also typings of M'. Restate subject reduction using the relation $\sqsubseteq$ and proof it.* (Solution p. 45) □

3.5 Normalization

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

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

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

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

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

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

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

Lemma 11 *For any type τ, the reduction of any term in $\mathcal{T}_\tau$ halts.*

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

Lemma 12 *If $\varnothing \vdash M : \tau$ and $M \longrightarrow M'$, then $M \in \mathcal{T}_\tau$ iff $M' \in \mathcal{T}_\tau$.*

(Proof p. 45)

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

Lemma 13 *If $\varnothing \vdash M : \tau$, then $M \in \mathcal{T}_\tau$.*

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

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

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

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

Case M is $M_1\ M_2$: By inversion of the typing judgment (1), we have $\Gamma \vdash M_1 : \tau_2 \to \tau$ (**4**) and $\Gamma \vdash M_2 : \tau_2$ (**5**) for some type τ_2. Therefore, by induction hypothesis applied to (4) and (5), we have $\theta M_1 \in \mathcal{T}_{\tau_2\to\tau}$ and $\theta M_2 \in \mathcal{T}_{\tau_2}$. Thus, by definition of $\mathcal{T}_\tau$, we have$(\theta M_1)\ (\theta M_2) \in \mathcal{T}_\tau$; that is, $\theta M \in \mathcal{T}_\tau$.

Case M is $\lambda x{:}\tau_1.\, M_2$: By inversion of the typing judgment (1), we have $\Gamma, x : \tau_1 \vdash M_2 : \tau_2$ (**6**) where $\tau_1 \to \tau_2$ is τ (**7**). Since M is a value, it is terminating. Hence, to ensure (3), it suffices to show that the application of θM to any M_1 in $\mathcal{T}_{\tau_1}$ is in $\mathcal{T}_{\tau_2}$ (**8**). Let $M_1 \in \mathcal{T}_{\tau_1}$. By definition of $\mathcal{T}_{\tau_1}$, the term M_1 reduces to some value V, which by subject reduction has type τ_1, and so is in $\mathcal{T}_{\tau_1}$ (**9**). We have:

$$
\begin{array}{rcll}
(\theta M)\ M_1 & \triangleq & (\theta(\lambda x{:}\tau_1.\, M_2))\ M_1 & \text{by definition of } M \\
& = & (\lambda x{:}\tau_1.\, \theta M_2)\ M_1 & \text{choose } x \mathrel{\#} \vec{x} \\
& \longrightarrow^* & (\lambda x{:}\tau_1.\, \theta M_2)\ V & \text{by (9)} \\
& \longrightarrow & [x \mapsto V](\theta M_2) & \text{by } (\beta) \\
& = & ([x \mapsto V]\theta) M_2 & \\
& \in & \mathcal{T}_{\tau_2} & \text{by I.H.}
\end{array}
$$

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

3.6 Simple extensions

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

3.6.1 Unit

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

$$\tau ::= \ldots \mid \mathsf{unit} \qquad V ::= \ldots \mid () \qquad M ::= \ldots \mid ()$$

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

$$\textsc{Unit}\quad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

Exercise 13 *Check that type soundness is preserved.* *(Solution p. 46)* □

Notice that the classification Lemma is no more degenerate.

3.6.2 Boolean

$$V ::= \ldots \mid \mathsf{true} \mid \mathsf{false} \qquad M ::= \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$$

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

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

In particular, $\mathsf{if}\ V\ \mathsf{then}\ E\ \mathsf{else}\ M$ or $\mathsf{if}\ V\ \mathsf{then}\ E\ \mathsf{else}\ M$ 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:

$$\mathsf{if\ true\ then}\ M_1\ \mathsf{else}\ M_2 \longrightarrow M_1 \qquad \mathsf{if\ false\ then}\ M_1\ \mathsf{else}\ M_2 \longrightarrow M_2$$

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

$$\tau ::= \ldots \mid \mathsf{bool}$$

The new typing rules are:

$$\textsc{True}\quad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{False}\quad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \textsc{IfThenElse}\quad \frac{\Gamma \vdash M_0 : \mathsf{bool} \qquad \Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash \mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \tau}$$

Exercise 14 *Given the new cases for the classification lemma (without proving them). Check that progress is preserved.* (Solution p. 46) □

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

3.6.3 Pairs

To extend the simply-typed λ-calculus with pairs, we extend values, expressions, and evaluation contexts as follows:

$$\begin{array}{rcl} i & ::= & 1 \mid 2 \\ M & ::= & \ldots \mid (M, M) \mid \mathsf{proj}_i\ M \end{array} \qquad \begin{array}{rcl} V & ::= & \ldots \mid (V, V) \\ E & ::= & \ldots \mid ([], M) \mid (V, []) \mid \mathsf{proj}_i\ [] \end{array}$$

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 inlinied i):

$$\mathsf{proj}_i\ (V_1, V_2) \longrightarrow V_i$$

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

$$\tau ::= \ldots \mid \tau \times \tau \qquad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2}\ \textsc{Pair} \qquad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{proj}_i\ M : \tau_i}\ \textsc{Proj}$$

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

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

3.6.4 Sums

Values, expressions, evaluation contexts are extended:

$$M \ ::= \ \ldots \mid \mathsf{inj}_i\ M \mid \mathsf{case}\ M\ \mathsf{of}\ V \diamond V \qquad \begin{array}{rcl} V & ::= & \ldots \mid \mathsf{inj}_i\ V \\ E & ::= & \ldots \mid \mathsf{inj}_i\ [] \mid \mathsf{case}\ []\ \mathsf{of}\ V \diamond V \end{array}$$

A new reduction rule is introduced:

$$\mathsf{case}\ \mathsf{inj}_i\ V\ \mathsf{of}\ V_1 \diamond V_2 \longrightarrow V_i\ V$$

Sum types are added to classify sums:

$$\tau ::= \ldots \mid \tau + \tau$$

Two new typing rules are introduced:

$$\textsc{Inj}\quad \frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \mathsf{inj}_i\ M : \tau_1 + \tau_2} \qquad \textsc{Case}\quad \frac{\Gamma \vdash M : \tau_1 + \tau_2 \qquad \Gamma \vdash V_1 : \tau_1 \to \tau \qquad \Gamma \vdash V_2 : \tau_2 \to \tau}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ V_1 \diamond V_2 : \tau}$$

Notice A property of the simply-typed λ-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:

$$V ::= \ldots \mid \mathsf{inj}_i\ V\ \mathsf{as}\ \tau$$

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 choosen for meta-theoretical studies.

Exercise 18 *Describe the extension with the option type.* □

3.6.5 Modularity of extensions

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

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

Then, in each case,

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

Moreover, the extensions are independent: they can be added to the λ-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 *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 λ-calculus with constants, for which type soundness can be established uniformly under reasonable assumptions relating the typing rules and reduction rules for constants. This is the approach that we will follow in the next chapter (§4).

3.6.6 Recursive functions

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

$$V ::= \ldots \mid \mu f{:}\tau.\, \lambda x.M \qquad\qquad M ::= \ldots \mid \mu f{:}\tau.\, \lambda x.M$$

A new reduction rule is introduced to unfold recursive calls:

$$(\mu f{:}\tau.\, \lambda x.M)\ V \longrightarrow [f \mapsto \mu f{:}\tau.\, \lambda x.M][x \mapsto V]M$$

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

$$\textsc{FixAbs}\ \frac{\Gamma, f : \tau_1 \to \tau_2 \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2}{\Gamma \vdash \mu f{:}\tau_1 \to \tau_2.\, \lambda x.M : \tau_1 \to \tau_2}$$

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

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

3.6.7 A derived construct: let-bindings

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

$$\textsc{App}\ \frac{\textsc{Abs}\ \dfrac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M_2 : \tau_1 \to \tau_2} \qquad \Gamma \vdash M_1 : \tau_1}{\Gamma \vdash (\lambda x{:}\tau_1.\, M_2)\ M_1 : \tau_2}$$

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

$$\textsc{LetMono}\quad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x : \tau_1 = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

In the derived form $\mathsf{let}\ x : \tau_1 = M_1\ \mathsf{in}\ M_2$ 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 $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$ we could use the following primitive typing rule.

$$\textsc{LetMono}\quad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

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

Sequences The sequence "$M_1; M_2$" is a derived construct of let-bindings; it can be viewed as additional syntactic sugar that expands to $\mathsf{let}\ x : \mathsf{unit} = M_1\ \mathsf{in}\ M_2$ where $x \# M_2$.

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

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

$$\textsc{LetMono}\ \frac{\textsc{FixAbs}\ \dfrac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \mu f\!:\!\tau \to \tau_1.\, \lambda x.M_1 : \tau \to \tau_1} \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ f = \mu f\!:\!\tau \to \tau_2.\, \lambda x.M_1\ \mathsf{in}\ M_2 : \tau_2}$$

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

$$\textsc{LetRecMono}\quad \frac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1 \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathsf{let\ rec}\ (f : \tau \to \tau_1)\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

3.7 Exceptions

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

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

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

3.7.1 Semantics

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

$$M \;::=\; \ldots \mid \mathsf{raise}\ M \mid \mathsf{try}\ M\ \mathsf{with}\ M \qquad\qquad E \;::=\; \ldots \mid \mathsf{raise}\ [\,] \mid \mathsf{try}\ [\,]\ \mathsf{with}\ M$$

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

$$\frac{\textsc{Raise}}{F[\mathsf{raise}\ V] \longrightarrow \mathsf{raise}\ V} \qquad \frac{\textsc{Handle-Val}}{\mathsf{try}\ V\ \mathsf{with}\ M \longrightarrow V} \qquad \frac{\textsc{Handle-Raise}}{\mathsf{try}\ \mathsf{raise}\ V\ \mathsf{with}\ M \longrightarrow M\ V}$$

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

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

Example Assume that K is $\lambda x.\, \lambda y.\, y$ and $M \longrightarrow V$. We have the following reduction:

$$\begin{array}{rll}
 & \mathsf{try}\ K\ (\mathsf{raise}\ M)\ \mathsf{with}\ \lambda x.\, x & \text{by } \textsc{Context} \\
\longrightarrow & \mathsf{try}\ K\ (\mathsf{raise}\ V)\ \mathsf{with}\ \lambda x.\, x & \text{by } \textsc{Raise} \\
\longrightarrow & \mathsf{try}\ \mathsf{raise}\ V\ \mathsf{with}\ \lambda x.\, x & \text{by } \textsc{Handle-Raise} \\
\longrightarrow & (\lambda x.\, x)\ V & \text{by } \beta \\
\longrightarrow & V &
\end{array}$$

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

$$\mathsf{try}\ K\ (\mathsf{raise}\ V)\ \mathsf{with}\ \lambda x.\, x \not\longrightarrow \mathsf{try}\ \lambda y.\, y\ \mathsf{with}\ \lambda x.\, x \longrightarrow \lambda y.\, y$$

3.7.2 Typing rules

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

$$\textsc{Raise}\ \frac{\Gamma \vdash M : \mathsf{exn}}{\Gamma \vdash \mathsf{raise}\ M : \tau} \qquad \textsc{Try}\ \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \mathsf{exn} \to \tau}{\Gamma \vdash \mathsf{try}\ M_1\ \mathsf{with}\ M_2 : \tau}$$

There are some subtleties: raise turns an expression of type exn into an exception. Consistently, the handler has type $\mathsf{exn} \to \tau$, since it receives as argument the value of type exn that has been raised. The expression raise M can have any type, since the current computation is aborted. In try M_1 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 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 exn for exceptions and allows adding new cases to the sum later on as needed. This is called an extensible datatype. As a counterpart checking for exceptions can't be exhaustive without a "catch all" branch, since further cases could always be added later. Notice that although new constructors may be added, the type of exception is fixed in the whole program, to exn. This is essential for type soundness, since the handling and raising of exceptions must agree globally on the type exn of exceptional values as it is not passed around.

Notice that exception constructors must have closed types since the type 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 $\varnothing \vdash M : \tau$ and $M \not\longrightarrow$, then M is either v or* raise *v for some value v.*

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

(Solution p. 47) □

3.7.3 Variations

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

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

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

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

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

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

Handle-Val'
$$\mathsf{try}\ V\ \mathsf{with}\ M \longrightarrow V$$

Handle-Raise'
$$\mathsf{try}\ \bar{F}[\mathsf{raise}\ V]\ \mathsf{with}\ M \longrightarrow M\ V$$

where $\bar{F}$ is sequence of F-contexts, *i.e.* a handler-free evaluation context of arbitrary depth. In this case, uncaught exceptions are of the form $\bar{F}[V]$. 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 $\mathsf{let}\ x = M_1\ \mathsf{with}\ M_2\ \mathsf{in}\ M_3$. 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 $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_3$; otherwise, *i.e.*, if it raises an exception $\mathsf{raise}\ V$, then the exception is handled by M_2, as if we had evaluated $\mathsf{try}\ M_1\ \mathsf{with}\ M_2$.

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

```
let rec read_config_in_path filename (dir :: dirs) →
  let fd = open_in (Filename.concat dir filename)
```

```
with Sys_error _ → read_config filename dirs in
read_config_from_fd fd
```

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

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

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

$$\mathsf{case}\ (\mathsf{try}\ \mathsf{Val}\ \ M_1\ \mathsf{with}\ \lambda y.\,\mathsf{Exc}\ \ y)\ \mathsf{of}\ (\mathsf{Val}\text{: }\lambda x.\,M_3 \diamond \mathsf{Exc}\text{: }M_2)$$

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

$$(\mathsf{try}\ \mathsf{let}\ x = M_1\ \mathsf{in}\ \lambda().\,M_3\ \mathsf{with}\ \lambda y.\,\lambda().\,M_2\ y)\ ()$$

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

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

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

3.8 References

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

$$M \ ::=\ \ldots \mid \mathsf{ref}\ M \mid M := M \mid\ !M \qquad\qquad E \ ::=\ \ldots \mid \mathsf{ref}\ [] \mid [] := M \mid V := [] \mid\ ![]$$

A reference allocation expression is not a value. Otherwise, by β-reduction, the program:

$$(\lambda x\!:\!\tau.\,(x := 1;\,!x))\ (\mathsf{ref}\ 3)$$

which intuitively should yield 1, would reduce to:

$$(\mathsf{ref}\ 3) := 1; !(\mathsf{ref}\ 3)$$

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

3.8.1 Language definition

Formally, we extend the simply-typed λ-calculus calculus with *memory locations*:

$$M ::= \ldots \mid \ell \qquad\qquad V ::= f \ldots \mid \ell$$

A memory location is just an atom (that is, a name). The value found at a location ℓ is obtained by indirection through a *memory* (or *store*). A memory μ 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 μ are in the domain of μ. Initially, the store is empty, and the term contains no locations, because, by convention, memory locations cannot appear in source programs. So, the invariant holds.

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

$$\begin{aligned} (\lambda x{:}\tau.\, M)\ V \,/\, \mu &\longrightarrow [x \mapsto V]M \,/\, \mu \\ E[M] \,/\, \mu &\longrightarrow E[M'] \,/\, \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\, \mu' \end{aligned}$$

Three new reduction rules are added:

$$\begin{aligned} \mathsf{ref}\ V \,/\, \mu &\longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathsf{dom}(\mu) \\ \ell := V \,/\, \mu &\longrightarrow () \,/\, \mu[\ell \mapsto V] \\ !\ell \,/\, \mu &\longrightarrow \mu(\ell) \,/\, \mu \end{aligned}$$

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

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

$$\tau ::= \ldots \mid \mathsf{ref}\ \tau$$

Three new typing rules are introduced:

$$\textsc{Ref}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{ref}\ M : \mathsf{ref}\ \tau} \qquad \textsc{Set}\ \frac{\Gamma \vdash M_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathsf{unit}} \qquad \textsc{Get}\ \frac{\Gamma \vdash M : \mathsf{ref}\ \tau}{\Gamma \vdash !M : \tau}$$

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

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

Then, typing judgments would have the form $\mu, \Gamma \vdash M : \tau$. typing judgments would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required. Moreover, if the value $\mu(\ell)$ happens to admit two distinct types τ_1 and τ_2, then ℓ admits types $\mathsf{ref}\ \tau_1$ and $\mathsf{ref}\ \tau_2$. So, one can write at type τ_1 and read at type τ_2: this rule is *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 *store typing* Σ, a finite mapping of locations to types.

Then, a location ℓ has type $\mathsf{ref}\ \tau$ "when the store typing Σ says so."

$$\textsc{Loc}\qquad \Sigma, \Gamma \vdash \ell : \mathsf{ref}\ \Sigma(\ell)$$

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

$$\textsc{Store}\quad \frac{\forall \ell \in \mathsf{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad\qquad \textsc{Config}\quad \frac{\Sigma, \varnothing \vdash M : \tau \qquad \vdash \mu : \Sigma}{\vdash M \,/\, \mu : \tau}$$

Remarks:

- 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.

3.8.2 Type soundness

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

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

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

Inlining Config, subject reduction can also be restated as:

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

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

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

Assuming compositionality, there exists τ' such that:

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

Then, by the induction hypothesis, there exists Σ' such that:

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

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

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

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

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

This allows establishing a strengthened version of compositionality:

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

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

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

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

By compositionality, there exists τ' such that:

$$\begin{array}{l} \Sigma, \varnothing \vdash M : \tau' \\ \forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \varnothing \vdash M' : \tau') \Rightarrow (\Sigma', \varnothing \vdash E[M'] : \tau') \end{array}$$

By the induction hypothesis, there exists Σ' such that:

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

The goal immediately follows.

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

3.8.3 Tracing effects with a monad

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

$$\begin{aligned}
\text{return} :&\ \alpha \to \mathsf{IO}\ \alpha \\
\text{bind} :&\ \mathsf{IO}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{IO}\ \beta \\
\text{main} :&\ \mathsf{IO}\ () \\
\text{newIORef} :&\ \alpha \to \mathsf{IO}\ (\mathsf{IORef}\ \alpha) \\
\text{readIORef} :&\ \mathsf{IORef}\ \alpha \to \mathsf{IO}\ \alpha \\
\text{writeIORef} :&\ \mathsf{IORef}\ \alpha \to \alpha \to \mathsf{IO}\ ()
\end{aligned}$$

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

3.8.4 Memory deallocation

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

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

Further reading

For a textbook introduction to λ-calculus and simple types, see Pierce (2002). For more details about syntactic type soundness proofs, see Wright and Felleisen (1994).

3.9 Ommitted proofs and answers to exercises

Solution of Exercise 8

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

Solution of Exercise 10

Case M 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 τ_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

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

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

Proof: Since $(\longrightarrow)$ is the smallest relation that satisfies rules Beta and Context, it suffices to show that $\sqsubseteq$ 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: Asume $M \sqsubseteq M'$. Let us show $E[M] \sqsubseteq E[M']$. Asume $\Gamma \vdash E[M] : \tau$. Then $\Gamma \vdash E[M'] : \tau$ follows by compositinality.

■

Proof of Lemma 12

By induction on the structure of the type τ.

Case τ is α: Then $\mathcal{T}_\tau$ is the set of terms that terminates. If $M \longrightarrow M'$, the termination of M, *i.e.* $M \in \mathcal{T}_\alpha$, is equivalent to the termination of M', *i.e.* $M' \in \mathcal{T}_\alpha$.

Case τ is $\tau_1 \to \tau_2$: Then $\mathcal{T}_\tau$ is the set of terms of type τ that terminate and also terminate when applied to any term M_1 of type τ_1. Assume $\varnothing \vdash M : \tau$ (**1**) and $M \longrightarrow M'$ (**2**). By subject reduction, we have $\varnothing \vdash M' : \tau$. Moreover, from (2), termination of M and termination of M' are equivalent. Therefore, it only remains to check that for any term M_1 of $\mathcal{T}_{\tau_1}$, $M\ M_1$ and $M'\ M_1$ are both in $\mathcal{T}_{\tau_2}$ or both outside of $\mathcal{T}_{\tau_2}$ (**3**). Let M_1 be in $\mathcal{T}_{\tau_1}$. We have $\varnothing \vdash M_1 : \tau$

and thus $\varnothing \vdash M\ M_1 : \tau_2$. We also have the call-by-value reduction $M\ M_1 \longrightarrow M'\ M_1$, Hence, (3) follows by induction hypothesis.

Solution of Exercise 13

Formally, we must revisit all the proofs. Auxialiary 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 Γ.

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

In the proof of progress, we have a new case for closed expressions, *i.e.* $()$, which happens to be a value, so it trivially satisfied the goal. Notice that although we do not need to invoke the classification for the new case of the $()$ expression, we still need to recheck the classification lemma, which is used in the case for application. The proof of the classification lemma is achieved by filling in the dots with a new case for a value of type 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 (**4**).

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

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

Case M is $\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.: By inversion of typing rules applied to (5), we have $\varnothing \vdash M_0 : \mathsf{bool}$, $\varnothing \vdash M_1 : \tau$, and $\varnothing \vdash M_2 : \tau$. If M_0 is a value, then, since it is of type bool, it must be true or false by (4), and in both cases, M reduces by either one of the two new rules. Otherwise, by induction hypothesis, M_0 myst be reducible, and so is M by rule Context since $\mathsf{if}\ []\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$ 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 interger constants $(\bar{n})_{n \in \boldsymbol{N}}$.

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

We add only evaluation contexts:

$$E \quad ::= \quad \ldots \mid [] + M \mid V + [] \mid [] * M \mid V * []$$

two reduction rules are:

$$\bar{n} + \bar{m} \longrightarrow \overline{n+m} \qquad\qquad \bar{n} * \bar{m} \longrightarrow \overline{n * m}$$

and the following typing rules:

$$\textsc{Int}\ \ \Gamma \vdash \bar{n} : \mathsf{int} \qquad \textsc{Plus}\ \frac{\Gamma \vdash M_1 : \mathsf{int} \qquad \Gamma \vdash M_2 : \mathsf{int}}{\Gamma \vdash M_1 + M_2 : \mathsf{int}} \qquad \textsc{Times}\ \frac{\Gamma \vdash M_1 : \mathsf{int} \qquad \Gamma \vdash M_2 : \mathsf{int}}{\Gamma \vdash M_1 \times M_2 : \mathsf{int}}$$

■

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:

$$\mathsf{proj}_i\ (V_1, V_2) \longrightarrow$$

We assume that $\Gamma \vdash \mathsf{proj}_i\ (V_1, V_2) : \tau$ (**7**). By inversion of typing of judgment, we know that the derivation of (7) ends with:

$$\textsc{Proj}\ \frac{\textsc{Pair}\ \dfrac{\Gamma \vdash V_1 : \tau_1\ (\mathbf{1}) \qquad \Gamma \vdash V_2 : \tau_2\ (\mathbf{2})}{\Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2}}{(7)}$$

with τ of the form $\tau_1 \to \tau_2$. We must show that $\Gamma \vdash V :_i \tau_i$ which is either one of the hypotheses (1) or (2). ■

Solution of Exercise 17

Just exchange M and V in the definition of evaluation contexts. This does not break sounness 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 exn equal to $\tau \to \tau$ where τ is $\mathsf{unit} \to \mathsf{unit}$. You may then define the inverse coercion functions between types $\tau \to \tau$ and τ:

$$\begin{array}{rcl} \mathsf{fold} & = & \lambda f \mathbin{:} \tau \to \tau.\, \lambda x \mathbin{:} \mathsf{unit}.\, \mathsf{let}\ z = \mathsf{raise}\ f\ \mathsf{in}\ () \\ \mathsf{unfold} & = & \lambda f \mathbin{:} \tau.\, \mathsf{try}\ \mathsf{let}\ z = f\ ()\ \mathsf{in}\ \lambda x \mathbin{:} \tau.\, x\ \mathsf{with}\ \lambda y \mathbin{:} \tau \to \tau.\, y \end{array}$$

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

Solution of Exercise 21

We need a new evaluation context:

$$E ::= \dots \mid \mathsf{let}\ x = E\ \mathsf{with}\ M_2\ \mathsf{in}\ M_3$$

and the following reduction rules:

Raise
$$F[\mathsf{raise}\ V] \longrightarrow \mathsf{raise}\ V$$

Handle-Val
$$\mathsf{let}\ x = V\ \mathsf{with}\ M_2\ \mathsf{in}\ M_3 \longrightarrow [x \mapsto V]M_3$$

Handle-Raise
$$\mathsf{let}\ x = \mathsf{raise}\ V\ \mathsf{with}\ M_2\ \mathsf{in}\ M_3 \longrightarrow M_2\ V$$

■

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.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3):81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Nikolaj Skallerud Bjørner. Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, 1994.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

▷ Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Crétin and Didier Rémy. Extending System F with Abstraction over Erasable Coercions. In *Proceedings of the 39th ACM Conference on Principles of Programming Languages*, January 2012.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

Jun Furuse. Extensional polymorphism by flow graph dispatching. In Ohori (2003), pages 376–393. ISBN 3-540-20536-5.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003b.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre 1, 2, ..., ω*. PhD thesis, Université Paris 7, September 1976.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169, New York, NY, USA, 1995a. ACM. ISBN 0-89791-719-7.

Mark P. Jones. Typing Haskell in Haskell. In *In Haskell Workshop*, 1999a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999b.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

▷ Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1, 2006.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

▷ Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

▷ Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6): 726–785, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.006.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, January 2006.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

▷ John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.

▷ John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

▷ Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

▷ Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

▷ Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. pages 233–244, 2002. doi: http://doi.acm.org/10.1145/565816.503294.

▷ Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

▷ Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

▷ Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Atsushi Ohori, editor. *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20536-5.

▷ Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

▷ Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254070.

▷ Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.

▷ Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

▷ Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA “Typage et Programmation”, December 2002.

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

François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

▷ Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994a.

▷ Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1994b.

▷ Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

▷ John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Science of Computer Programming*, 1994.

▷ Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.

▷ Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

▷ Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

▷ Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

▷ W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

▷ Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

▷ Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

▷ J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.

▷ J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

▷ Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8 (4):343–356, December 1995.

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.