Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of September 20, 2017

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; $(\mathsf{int} \to \mathsf{bool}) \to (\mathsf{list\ int} \to \mathsf{list\ int})$ describes a function that maps an integer predicate to an integer list transformer.

Types must be *sound*. That is, programs must behave as prescribed by their types. Hence, types must be *checked* and ill-typed programs must be rejected.

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

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

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

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

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

Today, the question is rather whether to use basic types (*e.g.* as in ML or System F) or 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:

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

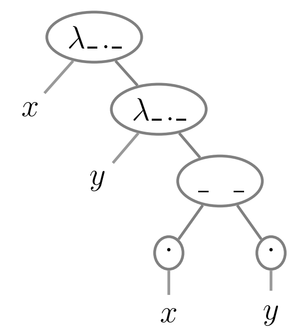
The expression $\lambda x.\, a$ *binds* variable x in a. We write $[x \mapsto a_0]a$ for the capture avoiding substitution of a_0 for x in a. Terms are considered equal up to the renaming of bound

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 polymorphic 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 HOAS (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 allow a *full* reduction, *i.e.* reduction can be performed 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 given a *weak* reduction strategy, *i.e.* reduction does not occur under abstractions. The main reason for this choice is simplicity and efficiency of 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 usually 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 full reduction, and then restricting the reduction paths to obtain the actual strategy. Full reduction may be used to model some program transformations, such as partial evaluation, that are performed at compile time. Another advantage of this two-step approach is that weak reduction strategies are a particular case of full reduction. Hence, (positive) properties can be established once

for all for full reduction and will also hold for weak reduction strategies, including both call-by-value and call-by-name.

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

2.2.2 Call-by-value semantics

We choose a *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-by-value 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:

$$\overset{\beta_v}{(\lambda x.\, a)\; v \longrightarrow [x \mapsto v]a} \qquad\qquad \overset{\text{CONTEXT}}{\dfrac{a \longrightarrow a'}{e[a] \longrightarrow e[a']}}$$

$[x \mapsto V]$ 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. 18)* □

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 x))
  | PApp (f, g)  → App (to_term f, to_term g)
  | PVar x → Var 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

XXX [*PED:* The evaluation context is not necessary. I am doing two changes at the same time... Fix/explain] The big-step semantics defines an evaluation relation

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

$$\textsc{Eval-Fun}\quad \mathcal{E} \vdash \lambda x.\, a \leadsto \lambda x.\, a \qquad \textsc{Eval-Var}\quad \frac{x \mapsto v \in \mathcal{E}}{\mathcal{E} \vdash x \leadsto v} \qquad \textsc{Eval-App}\quad \frac{\mathcal{E} \vdash a_1 \leadsto \lambda x.\, a \qquad \mathcal{E} \vdash a_2 \leadsto v_2 \qquad \mathcal{E}, x \mapsto v_2 \vdash a \leadsto v}{\mathcal{E} \vdash a_1\ a_2 \leadsto v}$$

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

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

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

Solution of Exercise 7

The definition contains an auto-application of a 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:

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

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 $\tau_1 \to \tau_2$.

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:

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

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:

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

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 $f : \tau_1 \to \tau_2; g : \tau_0 \to \tau_1; x : \tau_0$.

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

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 (direct simulation), and conversely, a reduction step after type erasure can also be traced back in the typed language as a reduction step between associated source terms (inverse simulation). Formally, this can be stated as follows:

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

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

Diagramatically, we have

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

Direct simulation | Inverse simulation

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

In our simple setting this is a one-to-one correspondence, and the proof is immediate and not very interesting. The proof will be done in the more general case of System F. In general (and this will be the case in System F) there may be reduction steps on source terms that involve only types and that have no counter-part on compiled terms. In this case we may split the reduction relation into $\longrightarrow_\iota$ that deals with those steps without counter-part on type-erasures and other steps such as $\longrightarrow_\beta$ that are reproduced type-erasures. The ι-reduction must be terminating (see the statement of bisimilation for System-F in §4.4.5).

Exercise 8 (Short, but difficult) *How would you write the two previous lemmas in the presence of ι-steps. What could happen if ι-reduction were not terminating?*

(Solution p. 44) □

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

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

3.4 Type soundness

Type soundness is often known as Milner's slogan "Well-typed expressions do not go wrong" What is a formal statement of this? By definition, a closed term M is *well-typed* if it admits some type 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:

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

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. 44)* □

Exercise 11 *Try to prove the unstrengthened version and see where you get stuck.* *(Solution p. 44)* □

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!

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

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

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

Soundness versus completeness Because the 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 prove it.* (Solution p. 44) □

3.5 Simple extensions

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

3.5.1 Unit

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

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

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

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

Exercise 13 *Check that type soundness is preserved.* (Solution p. 44) □

Notice that the classification Lemma is no longer degenerate.

3.5.2 Boolean

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

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:

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

Exercise 14 *Give the new cases for the classification lemma (without proving them). Check that progress is preserved.* *(Solution p. 45)* □

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. 45)* □

3.5.3 Pairs

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

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

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:

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

Exercise 16 *Check that subject reduction is preserved when adding pairs.*
(Solution p. 45) □

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. 46)* □

3.5.4 Sums

Values, expressions, evaluation contexts are extended:

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

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:

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

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.* □

3.5.5 Modularity of extensions

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

- a new type constructor, to classify values of a new shape;
- new expressions, to *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).

3.5.6 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:

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

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.

3.5.7 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:

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

This means that the following *derived rule* is sound and *complete* for let-bindings (a derived rule is a rule that abbreviates a prefix of a derivation tree):

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

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.

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

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:

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

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

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

3.6 Exceptions

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

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

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

3.6.1 Semantics

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

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

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:

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

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 $\lambda y.\,y$:

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

3.6.2 Typing rules

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

$$\text{RAISE}\ \frac{\Gamma \vdash M : \mathsf{exn}}{\Gamma \vdash \mathsf{raise}\ M : \tau} \qquad \text{TRY}\ \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \mathsf{exn} \to \tau}{\Gamma \vdash \mathsf{try}\ M_1\ \mathsf{with}\ M_2 : \tau}$$

There are some subtleties: raise turns an expression of type exn into an exception. Consistently, the handler has type 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. (Until recently, the type of exceptions was the only extensible datatypes in OCaml, but since verion 4.02, the user may define his own.)

As a counterpart checking for exceptions can't be exhaustive without a "catch all" branch, since further cases could always be added later. Notice that although new constructors may be added, the type of exception is fixed in the whole program, to 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. 46) □

3.6.3 Variations

Structured exceptions We have assumed that there is a unique exception, which could itself be a sum type. This simulates having multiple exceptions where each one is identified by a tag and may carry values of different types. However, having 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 $\bar{F}[\mathsf{raise}\ V]$. 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:

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

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. 46)* □

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

```
let rec read_config_in_path filename (dir :: dirs) →
  match open_in (Filename.concat dir filename) with
  | fd → read_config_from_fd fd
  | exception Sys_error _ → read_config filename dirs
```

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. (You may try first without using binding mixed with exceptions and then using it.) this construct.)* *(Solution p. 46)* □

3.7 References

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

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

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.

3.7.1 Language definition

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

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

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:

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

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[1] 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."

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

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

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

Remarks:

[1] This could happen, for example, in the presence of sum types (described in §3.5.4), when expressions do not have unique types any longer.

- This is an *inductive* definition. The store typing Σ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing is used only in the definition of a "well-typed configuration" and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.

3.7.2 Type soundness

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

Theorem 5 (Subject reduction) *Reduction preserves types: if $M \,/\, \mu \longrightarrow M' \,/\, \mu'$ 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 11 (Stability under memory allocation) *If $\Sigma \subseteq \Sigma'$ and $\Sigma, \Gamma \vdash M : \tau$, then $\Sigma', \Gamma \vdash M : \tau$.*

This allows establishing a strengthened version of compositionality:

Lemma 12 (Compositionality) *Assume $\Sigma, \varnothing \vdash E[M] : \tau$. 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.* □

3.7.3 Tracing effects with a monad

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

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

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

3.7.4 Memory deallocation

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

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

Further reading

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

3.8 Ommitted proofs and answers to exercises

Solution of Exercise 8

See the statement of bisimilation for System-F in §4.4.5, in particular lemmas 21 and **??**. ■

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 prove it as follows:

<u>Proof</u>: 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.

■

Solution of Exercise 13

Formally, we must revisit all the proofs. Auxiliary lemmas such as permutation and weakening still hold without any problem: in the proof by structural induction, there is a new case for unit expressions, which is proved by an application of the same rule, Unit but with possibly a different context 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 (**5**).

For the proof of progress, we assume that $\varnothing \vdash M : \tau$ (**6**) and show that M is either a value or reducible (**4??**) by structural induction on M. We have two new cases:

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

Case M is $\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.: By inversion of typing rules applied to (6), 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 (5), 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 $(\bar{n})_{n \in N}$.

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

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:

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

■

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 $\Gamma \vdash \mathsf{proj}_i\ (V_1, V_2) : \tau$ (**2??**). By inversion of typing of judgment, we know that the derivation of (2) ends with:

$$\textsc{Proj}\ \dfrac{\textsc{Pair}\ \dfrac{\Gamma \vdash V_1 : \tau_1\ \mathbf{(1)} \qquad \Gamma \vdash V_2 : \tau_2\ \mathbf{(3)}}{\Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2}}{(2)}$$

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 (3). ■

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:

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

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:

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

and the following reduction rules:

$$\textsc{Raise}\quad F[\mathsf{raise}\ V] \longrightarrow \mathsf{raise}\ V \qquad\qquad \textsc{Handle-Val}\quad \mathsf{let}\ x = V\ \mathsf{with}\ M_2\ \mathsf{in}\ M_3 \longrightarrow [x \mapsto V]M_3$$

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

■

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
```

This may also be written, more concisely:

```
let finalize f x g y =
  match f x with
  | result → g y; result
  | exception exn → g y; raise exn
```

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. ∎

Chapter 4

Polymorphism and System F

4.1 Polymorphism

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

$$\forall\alpha.\,(\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

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

$$(\mathsf{int} \to \mathsf{int} \to \mathsf{bool}) \to \mathsf{list\ int} \to \mathsf{list\ int} \qquad (\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}) \to \mathsf{list\ bool} \to \mathsf{list\ bool} \qquad \ldots$$

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

$$(\top \to \top \to \mathsf{bool}) \to \mathsf{list}\ \top \to \mathsf{list}\ \top$$

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

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

$$(\tau_1 \to \tau_2) \to (\tau_0 \to \tau_1) \to \tau_0 \to \tau_2$$

among which is

$$(\alpha_1 \to \alpha_2) \to (\alpha_0 \to \alpha_1) \to \alpha_0 \to \alpha_2$$

which is principal, as all other types can be recovered by instantiation of the variables. By

saying that this term admits the polymorphic type

$$\forall \alpha_1 \alpha_2. (\alpha_1 \to \alpha_2) \to (\alpha_0 \to \alpha_1) \to \alpha_0 \to \alpha_2$$

we make polymorphism *internal* to the type system.

Polymorphism is a step on the road towards *type abstraction.* Intuitively, if a function that sorts a list has polymorphic type

$$\forall \alpha. (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

then it *knows nothing* about α—it is *parametric* in α—so it must manipulate the list elements *abstractly:* it can copy them around, pass them as arguments to the comparison function, but it cannot directly inspect their structure. In short, within the code of the list sorting function, the variable α is an *abstract type.*

Parametricity In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it. For instance, the polymorphic type $\forall \alpha. \alpha \to \alpha$ has only one inhabitant, namely the identity. Similarly, the type of the list sorting function

$$\forall \alpha. (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

reveals a *"free theorem"* about its behavior! Basically, sorting commutes with (map f), provided f is order preserving. Note that there are many inhabitants of this type (e.g. a function that sorts in reverse order, or a function that removes duplicates) but they all satisfy this free theorem. This phenomenon was studied by Reynolds 1983 and by Wadler 1989; 2007, among others. An account based on an operational semantics is offered by Pitts 2000.

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

- With parametric polymorphism, a term can admit several types, all of which are *instances* of a common polymorphic type: $\mathsf{int} \to \mathsf{int}$, $\mathsf{bool} \to \mathsf{bool}$, ... and $\forall \alpha. \alpha \to \alpha$.

  With ad hoc polymorphism, a term can admit a collection of *unrelated* types: $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{float} \to \mathsf{float} \to \mathsf{float}$, ... but *not* $\forall \alpha. \alpha \to \alpha \to \alpha$.

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

  With ad hoc polymorphism, untyped programs do not have a semantics: *the meaning of a term can depend upon its type* (e.g. $2+2$), or, even worse, *upon its type derivation* (e.g. $\lambda x. \mathit{show}\ (\mathit{read}\ x)$).

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

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}\;\textsc{Tabs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}\;\textsc{Tapp}$$

Figure 4.1: Typing rules for System F.

By the first definition, Haskell's *type classes* (Hudak et al., 2007) are a form of (bounded) parametric polymorphism: terms have *principal (qualified) type schemes*, such as:

$$\forall\alpha.\,\mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

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

4.2 Polymorphic λ-calculus

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

4.2.1 Types and typing rules

Types of the simply-typed λ-calculus are extended with polymorphic types:

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

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

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

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

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

Type variables are explicitly bound and appear in type environments:

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

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

Well-formedness of environments and types may be defined (recursively) by inference rules (Rule WfEnvVar depends on well-formedness of types while Rule WfTypeVar depends on well-formedness of environments):

$$\textsc{WfEnvEmpty}\quad \vdash \varnothing \qquad \textsc{WfEnvVar}\quad \frac{\vdash \Gamma \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau} \qquad \textsc{WfEnvTvar}\quad \frac{\vdash \Gamma \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \alpha}$$

$$\textsc{WfTypeVar}\quad \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{WfTypeArrow}\quad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \textsc{WfTypeForall}\quad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\, \tau}$$

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

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

$$\textsc{Var}\quad \frac{x : \tau \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Tapp}\quad \frac{\Gamma \vdash M : \forall \alpha.\tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

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

4.2.2 Semantics

We need the following reduction for type abstraction:

$$(\Lambda \alpha.M)\ \tau \longrightarrow [\alpha \mapsto \tau]M \qquad (\iota)$$

Then, there is a choice regarding whether type abstraction should stop the evaluation, or let reduction proceed.

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

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

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

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

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

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

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

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

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

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

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

Accordingly, we only need a weaker version of ι-reduction:

$$(\Lambda\alpha.V)\ \tau \longrightarrow [\alpha \mapsto \tau]V \tag{ι_v}$$

We now have:

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

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

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

On the opposite the *type-erasing* semantics, has several advantages: it does not alter the semantics of untyped terms; it coincides with the semantics of ML—and, more generally,

with the semantics of most programming languages. It also exhibits difficulties when adding side effects while the type-passing semantics keeps them hidden.

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

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

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

4.2.3 Extended System F with datatypes

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

$$\begin{array}{rcl}
Pair & \triangleq & \Lambda\alpha_1.\Lambda\alpha_2.\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\,\Lambda\beta.\lambda y{:}\alpha_1 \to \alpha_2 \to \beta.\,y\;x_1\;x_2 \\
proj_i & \triangleq & \Lambda\alpha_1.\Lambda\alpha_2.\lambda y{:}\forall\beta.\,(\alpha_1 \to \alpha_2 \to \beta) \to \beta.\,y\;\alpha_i\;(\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\,x_i) \\
\lceil Pair \rceil & \triangleq & \lambda x_1.\,\lambda x_2.\,\lambda y.\,y\;x_1\;x_2 \\
\lceil proj_i \rceil & \triangleq & \lambda y.\,y\;(\lambda x_1.\,\lambda x_2.\,x_i)
\end{array}$$

Notice the use of first-class polymorphism in the definition of $proj_i$. This is general in the encoding of datatypes.

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

Unit, Pairs, Sums, *etc.* can also be added to System F as primitives. We can then proceed as for simply-typed λ-calculus. However, we may also take advantage of the expressive type system of System F to deal with such extensions in a more elegant way: thanks to polymorphism, we need not add new typing rules for each extension. We may instead add one typing rule for constants and parametrize the definition by an initial typing environment Δ for constants. This allows sharing the meta-theoretical developments between the different extensions.

Adding primitive pairs Let us first illustrate datatypes on an example, adding primitive pairs to System F. We will then generalize the presentation to parametrize the extension as suggested above.

We introduce a new type constructor $(\cdot \times \cdot)$ of arity 2 to classify pairs:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \tau \times \tau$$

Expressions are extended with a constructor $(\cdot, \cdot)$ and two destructors proj_1 and proj_2 with the respective signatures:

$$\begin{array}{ll} \mathit{Pair}: & \forall \alpha_1.\, \forall \alpha_2.\, \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ \mathit{proj}_i: & \forall \alpha_1.\, \forall \alpha_2.\, \alpha_1 \times \alpha_2 \to \alpha_i \end{array}$$

that forms the initial typing environment Δ. We need not add any new typing rule, but instead type programs in the initial environment Δ.

This allows for the formation of partial applications of constructors and destructors. Hence, values are extended as follows:

$$\begin{array}{lll} V & ::= & \ldots \mid \mathit{Pair} \mid \mathit{Pair}\ \tau \mid \mathit{Pair}\ \tau\ \tau \mid \mathit{Pair}\ \tau\ \tau\ V \mid \mathit{Pair}\ \tau\ \tau\ V\ V \\ & & \mid \mathit{proj}_i \mid \mathit{proj}_i\ \tau \mid \mathit{proj}_i\ \tau\ \tau \end{array}$$

We add the two following reduction rules:

$$\mathit{proj}_i\ \tau_1\ \tau_2\ (\mathit{Pair}\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Notice that, for well-typed programs, τ_i and τ_i' will always be equal, but the reduction will not check this at runtime. This could be enforced by replacing δ with the following rule:

$$\mathit{proj}_i\ \tau_1\ \tau_2\ (\mathit{Pair}\ \tau_1\ \tau_2\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair}')$$

The two semantics coincide on well-typed terms, but differ on ill-typed terms where δ_{pair}' may block when rule δ_{pair} would progress, ignoring type errors. Interestingly, using δ_{pair}' simplifies the proof obligation in subject reduction but introduces a more stronger proof obligation in progress.

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

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

of the extension then automatically follows.

We assume given a collection of constants, written with letter c, each of which given with a fix arity written $\mathsf{arity}\,(c)$. Constants must actually be partitioned into constructors (written C) and destructors (written d); moreover, we disallow nullary destructors[1].

Expressions are extended with constant expressions.

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

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

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

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

$$d\ \tau_1 \ldots\ \tau_j\ V_1 \ldots\ V_k \longrightarrow M \tag{δ_d}$$

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

$$\tau ::= \ldots \mid \mathsf{G}\ \tau_1 \ldots \tau_n$$

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

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

$$\frac{\overset{\textsc{Cst}}{c : \tau \in \Delta} \qquad \vdash \Gamma}{\Gamma \vdash c : \tau}$$

Taking typing constraints into account, we may give a more restrictive characterization of well-typed values: in the presentation above i is at most the number of quantified variables in the type scheme of the constructor, and whenever n is non zero, i is equal to this number. And similarly for destructors. For instance, if C is a constructor (respectively, d is a destructor) of arity q and of type $\forall\alpha_1 \ldots \alpha_p. \tau'_1 \to \ldots \tau'_q \to \tau$, then values will contain:

$$C \mid C\ \tau_1 \mid \ldots\ C\ \tau_1\ \ldots \tau_p \mid C\ \tau_1\ \ldots \tau_p\ V_1 \mid \ldots\ C\ \tau_1\ \ldots \tau_p\ V_1\ \ldots V_q$$

[1]Nullary polymorphic destructors introduce pathological cases to maintain the semantics type-erasing—for little benefit in return.

and

$$c \mid c\,\tau_1 \mid \ldots\; c\,\tau_1\;\ldots\tau_p \mid c\,\tau_1\;\ldots\tau_p\,V_1 \mid \ldots\; c\,\tau_1\;\ldots\tau_p\,V_1\;\ldots V_{q-1}$$

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

Definition 1 *δ-reduction is* sound *if it preserves typings and ensures progress for primitives. That is*

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

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

We will show below that soundness of δ-rules is sufficient to ensure soundness of the extension.

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

Exercise 24 (Pairs as constants) *Reformulate the extension of System F with pairs as constants. Check soundness of the δ-rules.* *(Solution p. 84)* □

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

(Solution p. 84) □

Exercise 26 (List) *1) Formulate the extension of System F with lists as constants. 2) Check that this extension is sound.* *(Solution p. 84)* □

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

However, we may introduce a fixpoint as a binary primitive with the following typing assumption:

$$\mathsf{fix} : \forall\alpha.\,\forall\beta.\,((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

and the reduction rule:

$$\mathsf{fix}\,\tau_1\,\tau_2\,V_1\,V_2 \longrightarrow V_1\,(\mathsf{fix}\,\tau_1\,\tau_2\,V_1)\,V_2 \qquad (\delta_{\mathsf{fix}})$$

It is straightforward to check the soundness of this extension: Progress is by construction, since fix does not destruct values. As for subject reduction, assume $\Gamma \vdash \mathsf{fix}\,\tau_1\,\tau_2\,V_1\,V_2 : \tau$. By inversion of typing rules, τ must be equal to τ_2, V_1 and V_2 must be of respective types $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ and τ_1 in the typing context Γ. We may then easily build a derivation of the judgment $\Gamma \vdash V_1\,(\mathsf{fix}\,\tau_1\,\tau_2\,V_1)\,V_2 : \tau$.

Exercise 27 (Recursion with datatypes) *In ML a one-constructor datatype can be used to emulate recursive types, namely a type* Any *such that a value of type* $\mathsf{any} \to \mathsf{any}$ *can be converted to a value of type* any*, and conversely. Give the definition in ML. Describe the extension as the addition of new constants. Verify the soundness of δ-rules.*

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

$$((\mathsf{any} \to \mathsf{any}) \to \mathsf{any} \to \mathsf{any}) \to \mathsf{any} \to \mathsf{any}$$

in ML without using `let rec` *or implicit recursive types (the −rectypes option). (See Exercise 7 for a definition of the fix-point in the λ-calculus or in ML with recursive types.)*

(Solution p. 85) □

4.3 Type soundness

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

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

Lemma 13 (Inversion of typing rules) *Assume* $\Gamma \vdash M : \tau$*.*

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

Lemma 14 (Permutation) *If* Γ *and* Γ' *are two well-formed permutations, then* $\Gamma \vdash M : \tau$ *iff* $\Gamma \vdash M : \tau$*.*

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

Lemma 15 (Weakening) *If* $\Gamma \vdash M : \tau$ *and* $\vdash \Gamma, \Gamma'$*, then* $\Gamma, \Gamma' \vdash M : \tau$*.*

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

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

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

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

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

Case M is $M_1\ M_2$ or $M_1\ \tau_1$: By inversion of typing, induction hypothesis applied to the premises, and APP or TAPP to conclude.

Lemma 16 (Expression substitution, strengthened)
If $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ and $\Gamma \vdash M_0 : \tau_0$ then $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$.

We have strengthened the lemma with an arbitrary context Γ' as for the simply-typed λ-calculus. We have also generalized the lemma with an arbitrary context Γ on the left and an arbitrary expression M, as this does not complicate the proof (and the stronger result will be used later). The proof is similar to the one for the simply-typed λ-calculus, with just a few more cases. (Details of the proof p. 85)

Exercise 28 *Write the details of the proof.* □

Lemma 17 (Type substitution, strengthened)
If $\Gamma, \alpha, \Gamma' \vdash M : \tau$ and $\Gamma \vdash \tau_0$ then $\Gamma, \theta\Gamma' \vdash \theta M : \theta\tau$ where θ is $[\alpha \mapsto \tau_0]$.

As for expression substitution, we have strengthened the lemma and generalized it using an arbitrary environment instead of the empty environment, as it does not complicate the proof, but yields a stronger result. This lemma resembles the one for expression substitutions.

However, the substitution must also apply to the environment Γ' and the result type τ since α may appear free in them.

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

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

Lemma 18 (Compositionality) *If $\Gamma \vdash E[M] : \tau$, then there exists a sequence of type variables $\vec{\alpha}$ and τ' such that $\Gamma, \vec{\alpha} \vdash M : \tau'$ and all M' verifying $\Gamma, \vec{\alpha} \vdash M' : \tau'$ also verify $\Gamma \vdash E[M'] : \tau$.*

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

Notice that M' is typechecked in the context Γ extended with $\vec{\alpha}$, since the hole in the context E may be under type abstractions. We use the notation $\vec{\alpha}$ for a (possibly empty) sequence of type variables.

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

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

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

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

$$\textsc{App}\ \dfrac{\textsc{Abs}\ \dfrac{\Gamma, x : \tau' \vdash M_1 : \tau\ \mathbf{(2)}}{\Gamma \vdash (\lambda x{:}\tau'.\, M_1) : \tau' \to \tau} \qquad \Gamma \vdash V : \tau'\ \mathbf{(3)}}{\Gamma \vdash (\lambda x{:}\tau'.\, M_1)\ V : \tau\ (1)}$$

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

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

Case $E[M_0] \longrightarrow E[M_0']$: The hypothesis is $M_0 \longrightarrow M_0'$. Assume $\Gamma \vdash E[M_0] : \tau$. By compositionality, there is some type τ_0 and type variables $\vec{\alpha}$ such that $\Gamma, \vec{\alpha} \vdash M_0 : \tau_0$ (**5**) and for all M_0' such that $\Gamma, \vec{\alpha} \vdash M_0' : \tau_0$, we have $\Gamma \vdash E[M_0'] : \tau$. Therefore it suffices to show $\Gamma, \vec{\alpha} \vdash M_0' : \tau_0$, which holds by induction hypothesis applied to (5).

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

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

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

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

The proof is similar to the one for simply-typed λ-calculus.

Progress is restated as follows:

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

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

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

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

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

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

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

Case M is $\Lambda\beta.M_1$: Then, $\vec{\alpha}, \beta \vdash M_1 : \tau_1$. Since type abstraction is an evaluation context M_1 is irreducible. Hence, by induction hypothesis, M_1 is a value and so is M.

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

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

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

4.4 Type erasing semantics

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

4.4.1 Implicitly-typed System F

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

We write $\lceil \mathsf{F} \rceil$ for the set of implicitly-typed terms and F for the set of explicitly-typed terms. We use letters a, v, and e to range over implicitly-typed terms, values, and evaluation contexts, reusing the same notations as for the untyped λ-calculus.

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

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

Remark 4 Notice that the explicit introduction of variable α in the premise of Rule TABS contains an implicit side condition $\alpha \# \Gamma$ due to the assumption on the formation of typing environments.

$$\text{IF-VAR}\quad \Gamma \vdash x : \Gamma(x) \qquad\qquad \text{IF-CST}\quad \Gamma \vdash c : \Delta(c) \qquad\qquad \text{IF-ABS}\quad \dfrac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad\qquad \text{IF-APP}\quad \dfrac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\text{IF-TABS}\quad \dfrac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau} \qquad\qquad \text{IF-TAPP}\quad \dfrac{\Gamma \vdash a : \forall \alpha.\tau}{\Gamma \vdash a : [\alpha \mapsto \tau_0]\tau}$$

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

In implicitly-typed System F, as in ML, the introduction of type variables in typing context is often left implicit. (In some extensions of System F, type variables may carry a kind or a bound and must be explicitly introduced.) If we chose to do so, we would need an explicit side-condition on Rule TABS as follows:

$$\text{IF-TABS-BIS}\quad \dfrac{\Gamma \vdash a : \tau \qquad \alpha \mathrel{\#} \Gamma}{\Gamma \vdash a : \forall \alpha.\tau}$$

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

$$\text{IF-TABS-BIS}\ \dfrac{\text{IF-ABS}\ \dfrac{\text{IF-TAPP}\ \dfrac{\text{BROKEN IF-TABS}\ \dfrac{\text{IF-VAR}\ \dfrac{}{x : \alpha_1 \vdash x : \alpha_1}}{x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1}}{x : \alpha_1 \vdash x : \alpha_2}}{\varnothing \vdash \lambda x.\, x : \alpha_1 \to \alpha_2}}{\varnothing \vdash \lambda x.\, x : \forall \alpha_1.\forall \alpha_2.\alpha_1 \to \alpha_2}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha_1, \alpha_1 \vdash x : \alpha_1}\ \text{BROKEN VAR}}{x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1}\ \text{BROKEN TABS}}{x : \alpha_1 \vdash x : \alpha_2}\ \text{TAPP}}{\varnothing \vdash \lambda x{:}\alpha_1.\, x : \alpha_1 \to \alpha_2}\ \text{ABS}}{\varnothing \vdash \Lambda \alpha_1.\Lambda \alpha_2.\lambda \alpha_1{:}x.\, x : \forall \alpha_1.\forall \alpha_2.\alpha_1 \to \alpha_2}\ \text{TABS}$$

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

$$\forall \alpha.(P \Rightarrow Q) \quad \equiv \quad P \Rightarrow (\forall \alpha.Q) \qquad\qquad \text{if } \alpha \mathrel{\#} P$$

which without the side condition is obviously wrong.

The next lemma, states that the two definitions of $\lceil \mathsf{F} \rceil$—or, equivalently, the two type systems for implicitly-typed System F and explicitly type System F—coincide. The proof is immediate.

Lemma 20 $\Gamma \vdash a : \tau$ *in implicitly-typed System* F *if and only if there exists an explicitly-typed expression* M *whose erasure is* a *such that* $\Gamma \vdash M : \tau$.

For example, consider the term a_0 in $\lceil \mathsf{F} \rceil$ equal to $\lambda fxy.\,(f\ x,\ f\ y)$. A version that carries explicit type abstractions and annotations is:

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

Unsurprisingly, this term admits the polymorphic type:

$$\tau_1 \triangleq \forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

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

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

This term admits the polymorphic type:

$$\tau_2 \triangleq \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

This begs the question: which of the two types τ_1 or τ_2 is more general? Type τ_1 requires the second and third arguments to admit a common type, while type τ_2 requires the first argument to be polymorphic.

Exercise 30 (Distrib pair, disjoint types) *Find two terms a_1 and a_2 such that a_1 has type τ_1 but not type τ_2, and conversely for a_2. (Just give the terms a_1 and a_2, you do not have to prove well-typedness or ill-typedness.)* (Solution p. 86) □

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

Intuitively, one may think semantically of (*i.e.* interpret) a closed type as the set of terms of that type, and of instance as inclusion between types. With such a view in mind then τ_1 and τ_2 are indeed incomparable. This does not imply that a_0 does not have a principal type: there could exist a type τ_0 that contains a_0 and that is included in the intersection of (the interpretations of) τ_1 and τ_2. Indeed, one can do so in a richer system, such as System F^ω.

Exercise 31 (Distrib pair in F^ω) *Only if you know System F^ω: find a type τ_0 for a_0 in System F^ω that is more general than both τ_1 and τ_2,* i.e. *from which τ_1 and τ_2 can be obtained by rule* Inst-Gen. (Solution p. 86) □

4.4.2 Type instance

To reason formally, we must first define what it means for τ_2 to be an *instance* of τ_1—or, equivalently, for τ_1 to be *more general* than τ_2. Several definitions are possible. In System F, *to be an instance* is usually defined by the rule:

$$\frac{\vec{\beta} \mathrel{\#} \forall\vec{\alpha}.\,\tau}{\forall\vec{\alpha}.\tau \le \forall\vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau}\ \textsc{Inst-Gen}$$

Notice that $\vec{\alpha}$ and $\vec{\beta}$ stands of (possibly empty) sequences of type variables. One can show that, if $\tau_1 \le \tau_2$, then any term that has type τ_1 has also type τ_2; that is, the following rule is *admissible*[2] in the implicitly-typed version:

$$\textsc{Sub}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \le \tau_2}{\Gamma \vdash a : \tau_2}$$

Perhaps surprisingly, the rule is not *derivable*[3] in our presentation of System F. Although, we have the following derivation,

$$\textsc{Gen}^*\ \dfrac{\textsc{Inst}^*\ \dfrac{\Gamma, \vec{\beta} \vdash a : \forall \vec{\alpha}.\tau}{\Gamma, \vec{\beta} \vdash a : [\vec{\alpha} \mapsto \vec{\tau}]\tau}}{\Gamma \vdash a : \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

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

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

We may also wonder what is the counter-part of the instance relation in explicitly-typed System F. Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \le \tau_2$. How can we see M with type τ_2? Since explicitly-typed terms have unique types, the term M of type τ_1 cannot itself also have type τ_2. However, we can wrap M with a *retyping context* that transforms a term of type τ_1 to one of type τ_2. Since $\tau_1 \le \tau_2$, the types τ_1 and τ_2 must be of the form $\forall \vec{\alpha}.\tau$ and $\forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau$ where $\vec{\beta} \mathrel{\#} \forall \vec{\alpha}.\tau$. *W.l.o.g*, we may assume that $\vec{\beta} \mathrel{\#} \Gamma$ (6), as it may always be satisfied up to a renaming of bound variables $\vec{\beta}$. Then, we have the pseudo-derivation on the left-hand side (where the weakening lemma is used as a pseudo-typing rule Weakening), which can be abbreviated by the admissible typing rule Sub given on the right-hand side.

$$\left.\textsc{Tabs}^*\ \dfrac{\textsc{Tapp}^*\ \dfrac{\textsc{Weakening}\ \dfrac{\Gamma \vdash M : \forall \vec{\alpha}.\tau \qquad (\mathbf{6})\ \dfrac{\vec{\beta} \mathrel{\#} \forall \vec{\alpha}.\tau}{\vec{\beta} \mathrel{\#} \Gamma}}{\Gamma, \vec{\beta} \vdash M : \forall \vec{\alpha}.\tau}}{\Gamma, \vec{\beta} \vdash M\ \vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}}{\Gamma \vdash \Lambda \vec{\beta}.M\ \vec{\tau} : \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau}\right\}$$

Admissible rule:

$$\textsc{Sub}\ \dfrac{\Gamma \vdash M : \forall \vec{\alpha}.\tau \qquad \vec{\beta} \mathrel{\#} \forall \vec{\alpha}.\tau}{\Gamma \vdash \Lambda \vec{\beta}.M\ \vec{\tau} : \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

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

[3] A rule is *derivable* if it can be replaced by a sub-derivation tree with the same premises and conclusion.

In F, we rather write subtyping as a judgment $\Gamma \vdash \tau_1 \leq \tau_2$ instead of the binary relation $\tau_1 \leq \tau_2$ to also mean $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ and so simultaneously keep track of the well-formedness of types.

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

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

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

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

We can now give another equivalent definition of subtyping, based on retyping contexts: $\Gamma \vdash \tau_1 \leq \tau_2$ *if and only if there exists a retyping context* $\mathcal{R}$ *such that* $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$*.*

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

4.4.3 Type containment in System F_η

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

$$\textsc{Inst-Gen}\quad \frac{\vec{\beta} \# \forall\vec{\alpha}.\tau}{\forall\vec{\alpha}.\,\tau \leq \forall\vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau} \qquad \textsc{Distributivity}\quad \forall\alpha.\,(\tau_1 \to \tau_2) \leq (\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2) \qquad \textsc{Congruence-}\!\to\quad \frac{\tau_2 \leq \tau_1 \qquad \tau_1' \leq \tau_2'}{\tau_1 \to \tau_1' \leq \tau_2 \to \tau_2'}$$

$$\textsc{Congruence-}\forall\quad \frac{\tau_1 \leq \tau_2}{\forall\alpha.\,\tau_1 \leq \forall\alpha.\,\tau_2} \qquad \textsc{Transitivity}\quad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

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

Exercise 32 *1) Show that $\forall\alpha.\,\tau \equiv \tau$ whenever $\alpha \notin \mathsf{ftv}(\tau)$. 2) Show that rule* Distributivity *can be replaced by the weaker rule:*

$$\frac{\alpha \notin \mathsf{ftv}(\tau_1)}{\forall\alpha.\,(\tau_1 \to \tau_2) \le \tau_1 \to (\forall\alpha.\,\tau_2)}\;\textsc{Distrib-Right}$$

(Solution p. 87) □

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

$$\tau_{id} \quad \le \quad \left(\begin{array}{lcl} (\forall\alpha.\,\alpha) \to (\forall\alpha.\,\alpha) & \le & (\forall\alpha.\,\alpha) \to \tau_{id} \\ \forall\beta.\,(\beta \to \beta) \to (\beta \to \beta) & \le & \forall\beta.\,\tau_{id} \to (\beta \to \beta) \end{array}\right) \quad \le \quad \forall\beta.\,(\forall\alpha.\,\alpha) \to (\beta \to \beta)$$

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

$$ch \quad \triangleq \quad \lambda x.\,\lambda y.\,\mathsf{if}\ M\ \mathsf{then}\ x\ \mathsf{else}\ y \quad : \quad \forall\beta.\,\beta \to \beta \to \beta$$

Still, many expressions do not have most general types in System F_η. To see the difficulty, consider the application *chid* of *ch* to *id*. How can it be typed? If we keep *id* polymorphic, then *chid* has type $(\forall\alpha.\,\alpha \to \alpha) \to (\forall\alpha.\,\alpha \to \alpha)$, say τ_1; if, on the opposite, we instantiate *id*, then *chid* has type $\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$, say τ_2—as in ML where type schemes are automatically instantiated when used. These two types are incomparable in System F. Although, we have $\tau_1 \le \tau_2$ in System F_η (as witnessed by the coercion context $\lambda x{:}\forall\alpha.\,\alpha \to \alpha.\,\Lambda\alpha.([\tau_2]\ \alpha)\ (x\ \alpha))$ and can thus give *chid* the type τ_2 and still used it at type τ_1, this is more by chance than the general case: If we replace *ch* by ch_3, which chooses between three arguments, then ch_3 *id* does not have a principal type in System F_η.

System F_η increases the expressiveness of System F by enriching its type instance relation— without modifying the language of types (and other typing rules than Sub).

To obtain even more principal types, Le Botlan and Rémy (2009) have suggested that the language of types should be enriched with a new form of quantification $\forall\alpha \ge \tau_1.\,\tau_2$ to mean, intuitively, the set of types $[\alpha \mapsto \tau]\tau_2$ when τ ranges over the set of instances of τ_1. This internalizes the instance relation within the language of types. This allows to give *chid* the type $\forall(\beta \ge \forall\alpha.\,\alpha \to \alpha).\,\beta \to \beta$ and recovering $(\forall\alpha.\,\alpha \to \alpha) \to (\forall\alpha.\,\alpha \to \alpha)$ and $\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$ by choosing particular instances of $\forall\alpha.\,\alpha \to \alpha$ for β. By contrast with System F_η, this approach also works for the more general example of ch_3 *id*.

The language MLF has been design for partial type reconstruction where programs are partially annotated. The user need only to provide the types of parameters of functions

that are used polymorphically. The type systems is setup to implicitly use available polymorphism but never guess polymorphism. Available polymorphism comes either from type generalization as in ML or from user-provided type annotations. Every expression has a principal type—according to the given type annotations. See (Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008) for details.

4.4.4 A definition of principal typings

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

> *Every well-typed term M admits a principal typing – one whose* instances *are exactly the typings of M.*

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

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

> *A typing θ_1 is more general than a typing θ_2 if and only if every term that admits θ_1 admits θ_2 as well.*

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

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

There are still a few positive results. Some systems of *intersection types* have principal typings (Wells, 2002) – but they are very complex and have yet to see a practical application.

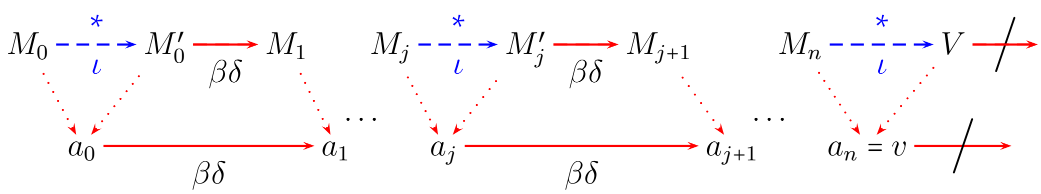
A weaker property is to have *principal types.* Given an environment Γ and an expression M is there a type τ for M in Γ such that all other types of M in Γ are instances of τ. Damas and Milner's type system (coming up next) does not have *principal typings* but it has *principal types* and *decidable type inference.*

4.4.5 Type soundness for implicitly-typed System F

Subject reduction and progress imply the soundness of the *explicitly*-typed version of *System* F. What about the *implicitly*-typed version? Can we reuse the soundness proof for the explicitly-typed version? Can we pullback subject reduction and progress from F to $\lceil \mathsf{F} \rceil$?

For progress, given a well-typed term a in $\lceil \mathsf{F} \rceil$, can we find a term M in F whose erasure is a and such that M is a value or reduces, and so conclude that a is a value or reduces? For subject reduction, given a term a_1 of type τ in $\lceil \mathsf{F} \rceil$ that reduces to a_2, can we find a term M_1 in F whose erasure is a_1 and show that M_1 reduces to a term M_2 whose erasure is a_2 to conclude that the type of a_2 is the type of a_1? In both cases, this reasoning requires a *type-erasing* semantics. We claimed that the explicitly-typed System F has an erasing semantics. We now verify it.

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



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

Lemma 21 (Direct simulation) *The reduction in* F *is simulated in* $\lceil \mathsf{F} \rceil$ *up to* ι*-steps. Assume* $\Gamma \vdash M : \tau$*. Then:*
1) If $M \longrightarrow_\iota M'$*, then* $\lceil M \rceil = \lceil M' \rceil$
2) If $M \longrightarrow_{\beta\delta} M'$*, then* $\lceil M \rceil \longrightarrow_{\beta\delta} \lceil M' \rceil$

The inverse direction is more delicate to state, since type erasure is not bijective: there are usually many expressions of F whose type erasure is a given expression in $\lceil \mathsf{F} \rceil$.

Lemma 22 (Inverse simulation) *Assume* $\Gamma \vdash M : \tau$ *and* $\lceil M \rceil \longrightarrow a$*. Then, there exists a term* M' *such that* $M \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M'$ *and* $\lceil M' \rceil = a$*.*

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

However, keeping δ-reduction abstract is preferable to avoid repeating the same reasoning many times. Then, we must assume that it is such that type erasure establishes a bisimulation for δ-reduction taken alone.

Assumption on δ. We assume that for any explicitly-typed term M of the form $d\,\tau_1 \ldots\, \tau_j\, V_1 \ldots\, V_k$ such that $\Gamma \vdash M : \tau$, both of the following properties hold:

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

(Inverse bisimulation) If $\lceil M \rceil \longrightarrow_\delta a$, then there exists M' such that $M \longrightarrow_\delta M'$ and a is the type-erasure of M'.

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

Exercise 33 *Consider the explicitly-typed System F with pairs of the exercise 24 (p. 57). Add pairs in the untyped λ-calculus. Show that δ-reduction in the untyped λ-calculus is larger than the image of the δ-reduction in the explicitly-typed calculus. Verify that type erasure is a bisimulation for δ-reduction.* *(Solution p. 87)* □

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

The inverse simulation is slightly more delicate because there may be many antecedents of a given type erasure. We use a few easy helper lemmas to keep the proof clearer.

Lemma 23

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

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

3) *If $\mathcal{R}[M]$ is in ι-normal form, then $\mathcal{R}$ is of the form $\Lambda\vec{\alpha}.[]\,\vec{\tau}$.*

The main helper lemma is :

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

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

The proof is by an induction on M.

Lemma 25 (Inversion of type erasure for well-typed values) *Assume $\Gamma \vdash M : \tau$ and M is ι-normal. If $\lceil M \rceil$ is a value v, then M is a value V. Moreover,*

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

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

Corollary 26 *Let M be a well-typed term in ι-normal whose erasure is a*

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

(Proof p. 88)

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

Termination of ι-reduction follows indirectly from the termination of reduction in System F. Its direct proof is also immediate, as ι-reduction strictly decreases the number of type abstractions.

Proof (inverse simulation): The proof is by induction on the reduction of $\lceil M \rceil$. We assume M is in ι-normal form.

Case $\lceil M \rceil$ is $(\lambda x.\, a_1)\; v$: By Corollary 26, M is of the form $\mathcal{R}[(\lambda x : \tau_1.\, M_1)\; V]$. Since $\mathcal{R}$ is an evaluation context, M reduces to $\mathcal{R}[[x \mapsto V]M_1]$ whose erasure is $[x \mapsto v]a_1$, *i.e.* a.

Case $\lceil M \rceil$ is $e[a_1]$ and $a_1 \longrightarrow a_2$: By Lemma 23, M is of the form $\bar{E}[M_1]$ where $\lceil \bar{E} \rceil$ is $\bar{e}$ and $\lceil M_1 \rceil$ is a_1. By compositionality (Lemma 18), M_1 is well-typed. Since M is ι-normal and $\bar{E}$ is an evaluation, M_1 is also ι-normal. By induction hypothesis, M_1 reduces in one $\beta\delta$ step to a term M_2 whose erasure is a_2. Hence, by Context, M reduces in one $\beta\delta$-step to the term $\bar{E}[M_2]$ whose erasure is $\bar{e}[a_2]$, *i.e.* a.

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

We may now easily transpose subject reduction and progress from the implicitly-typed version to the implicitly-typed version of System F.

Theorem 12 (Type soundness for implicitly-typed System F)
Progress and subject reduction holds in implicitly-typed System F.

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

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

Subject reduction: Assume $a_1 \longrightarrow a_2$. By inverse simulation (Lemma 22), there exists a term M_2 such that $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ and $\lceil M_2 \rceil$ is a_2. By subject reduction in F, we have $\Gamma \vdash M_2 : \tau$. By Lemma 20, we have $\Gamma \vdash a_2 : \tau$, as expected.

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

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

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

4.5 Polymorphism and references

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

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type.* Otherwise, if a location had two types ref τ_1 and ref τ_2, one could store a value of type τ_1 and read back a value of type τ_2. Hence, it should also be unsound if a location could have type $\forall\alpha.\, \mathsf{ref}\ \tau$ (where α appears in τ) as

it could then be specialized to both types $\mathsf{ref}\ [\alpha \mapsto \tau_1]\tau$ and $\mathsf{ref}\ [\alpha \mapsto \tau_2]\tau$. By contrast, a location ℓ can have type $\mathsf{ref}\ (\forall\alpha.\,\tau)$: this says that ℓ stores values of polymorphic type $\forall\alpha.\,\tau$, but ℓ, as a value, is viewed with the monomorphic type $\mathsf{ref}\ (\forall\alpha.\,\tau)$.

4.5.1 A counter example

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

$\mathsf{let}\ y : \forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha) =$	(2) Abstracts α and binds ℓ to y of type $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$
$\quad\Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\,z)$	(1) Creates and returns a location ℓ of type $\mathsf{ref}\ (\alpha \to \alpha)$
in	bound to the identity function $\lambda z{:}\alpha.\,z$ of type $\alpha \to \alpha$
$(y\ \mathsf{bool}) := (\mathsf{bool} \to \mathsf{bool})\ \mathit{not};$	(3) Writes the location at type $\mathsf{bool} \to \mathsf{bool}$
$!(\mathsf{int} \to \mathsf{int})\ (y\ (\mathsf{int}))\ 1 \quad / \quad \varnothing$	(4) Reads it back at type $\mathsf{int} \to \mathsf{int}$

$$\xrightarrow{*} \quad \mathit{not}\ 1 \quad / \quad \ell \mapsto \mathit{not}$$

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

$$\textsc{Context}\ \frac{\mathsf{ref}\ \tau\ V \,/\, \varnothing \longrightarrow \ell \,/\, \ell \mapsto V}{\Lambda\alpha.\mathsf{ref}\ \tau\ V \,/\, \varnothing \longrightarrow \Lambda\alpha.\ell \,/\, \ell \mapsto V}$$

While we have

$$\alpha \vdash \mathsf{ref}\ \tau\ V \,/\, \varnothing : \mathsf{ref}\ \tau \qquad \text{and} \qquad \alpha \vdash \ell \,/\, \ell \mapsto V : \mathsf{ref}\ \tau$$

We have

$$\vdash \Lambda\alpha.\mathsf{ref}\ \tau\ V \,/\, \varnothing : \forall\alpha.\,\mathsf{ref}\ \tau \qquad \text{but not} \qquad \vdash \Lambda\alpha.\ell \,/\, \ell \mapsto V : \forall\alpha.\,\mathsf{ref}\ \tau$$

Hence, the context case of subject reduction breaks.

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

$$\textsc{Tabs}\ \frac{\Sigma, \alpha \vdash \ell : \mathsf{ref}\ \tau}{\Sigma \vdash \Lambda\alpha.\ell : \forall\alpha.\,\mathsf{ref}\ \tau}$$

However, the typing context Σ, α is ill-formed as α appears free in Σ. Instead, a well-formed premise should bind α earlier as in $\alpha, \Sigma \vdash \ell : \mathsf{ref}\ \tau$, but then, Rule Tabs cannot be applied.

By contrast, the expression $\mathsf{ref}\ \tau\ V$ is pure, so Σ may be empty:

$$\textsc{Tabs}\ \frac{\alpha \vdash \mathsf{ref}\ \tau\ V : \mathsf{ref}\ \tau}{\varnothing \vdash \Lambda\alpha.\mathsf{ref}\ \tau\ V : \forall\alpha.\,\mathsf{ref}\ \tau}$$

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

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

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

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

$$\text{TAbs}\quad \frac{\Gamma, \alpha \vdash U : \tau}{\Gamma \vdash \Lambda\alpha.U : \forall\alpha.\tau} \qquad\qquad \text{Value forms:}\quad U ::= x \mid V \mid \Lambda\alpha.U \mid U\ \tau$$

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

4.5.2 Internalizing configurations

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

$$\begin{array}{rllll} & \mathsf{let\ rec}\ \ m : \Sigma & = \mu & \mathsf{in}\ E[\mathsf{ref}\ \tau\ V] \\ \longrightarrow & \mathsf{let\ rec}\ \ m : \Sigma, \ell : \tau & = \mu, \ell \mapsto v & \mathsf{in}\ E[m.\ell] \end{array}$$

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

Let use clarify the typing rules for configurations:

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

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

Judgments are now of the form $\vec{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\vec{\alpha}, \Sigma, \Gamma$ as a whole typing context Γ'. For locations, we need a new context formation rule:

$$\textsc{WfEnvLoc}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \ell \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \ell : \tau}$$

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

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

$$\textsc{Loc}\quad \Gamma \vdash \ell : \mathsf{ref}\ \Gamma(\ell)$$

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

$$\mathsf{ref}\colon\ \forall \alpha.\, \alpha \to \mathsf{ref}\ \alpha \qquad (!)\ :\ \forall \alpha.\, \mathsf{ref}\ \alpha \to \alpha \qquad (:=)\ :\ \forall \alpha.\, \mathsf{ref}\ \alpha \to \alpha \to \mathsf{unit}$$

They are all destructors (event ref) with the obvious arities.

The δ-rules are adapted to carry explicit type parameters:

$$\mathsf{ref}\ \tau\ V \,/\, \mu \longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathsf{dom}(\mu)$$

$$\ell := (\tau)\ V \,/\, \mu \longrightarrow ()\,/\, \mu[\ell \mapsto V] \qquad\qquad !\tau\ \ell \,/\, \mu \longrightarrow \mu(\ell) \,/\, \mu$$

Type soundness can now be stated as

Lemma 27 *δ-rules preserve well-typedness of closed configurations.*

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

Lemma 28 *A well-typed closed configuration M/μ where M is a full application of constants* ref*,* $(!)$*, and* $(:=)$ *to types and values can always be reduced.*

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

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

$$\begin{array}{l} \mathsf{let}\ y : \forall \alpha.\, \mathsf{ref}\ (\alpha \to \alpha) = \Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)\ \mathsf{in} \\ \quad (y\ \mathsf{bool}) := (\mathsf{bool} \to \mathsf{bool})\ not; \\ \quad !(\mathsf{int} \to \mathsf{int})\ (y\ (\mathsf{int}))1 \end{array}$$

Indeed, $\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)$ is not a value, but the application of a unary destructor to a value, so the expression $\Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)$ is not allowed.

Consequences With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references. This style of introducing references in System F (or in ML) is *not a conservative extension.*

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

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

- $\Lambda\alpha.((\lambda x{:}\tau.\,U)\;U)$ because the inner expression is non-expansive;
- $\Lambda\alpha.(\mathsf{let}\;x : \tau = U\;\mathsf{in}\;U)$, which is its syntactic sugar;
- $\mathsf{let}\;x : \forall\alpha.\,\mathsf{list}\;\alpha = \Lambda\alpha.(M_1\;M_2)\;\mathsf{in}\;M$ because α appears only positively in the type of $eappM_1M_2$.

OCaml implements both refinements.

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

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

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

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

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

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

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

4.6 Damas and Milner's type system

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

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

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

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad\longrightarrow\quad a_1; [x \mapsto a_1]a_2$$

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

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad\longrightarrow\quad [x \mapsto a_1]a_2$$

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

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

4.6.1 Definition

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

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

The *let* construct is no longer sugar for a β-redex but a primitive form that will be typed especially.

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

$$\tau ::= \alpha \mid \tau \to \tau \mid \ldots \qquad\qquad \sigma ::= \tau \mid \forall \alpha.\, \sigma$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{iml-Var} \qquad \frac{}{\Gamma \vdash c : \Delta(c)}\ \textsc{iml-Cst} \qquad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{iml-Abs} \qquad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{iml-App}$$

$$\frac{\Gamma \vdash a_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash a_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \sigma_2}\ \textsc{iml-Let} \qquad \frac{\Gamma, \alpha \vdash a : \sigma}{\Gamma \vdash a : \forall \alpha.\, \sigma}\ \textsc{iml-Gen} \qquad \frac{\Gamma \vdash a : \forall \alpha.\, \sigma}{\Gamma \vdash a : [\alpha \mapsto \tau]\sigma}\ \textsc{iml-Inst}$$

Figure 4.3: Typing rules for ML

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

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

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha, z : \alpha \vdash z : \alpha}\ \textsc{Var}}{\alpha \vdash \lambda z.\, z : \alpha \to \alpha}\ \textsc{Abs}}{\varnothing \vdash \lambda z.\, z : \forall \alpha.\, \alpha \to \alpha}\ \textsc{Gen} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall \alpha.\, \alpha \to \alpha}\ \textsc{Var}}{\Gamma \vdash f : \mathsf{int} \to \mathsf{int}}\ \textsc{Inst}}{\Gamma \vdash f\ 0 : \mathsf{int}}\ \textsc{App} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall \alpha.\, \alpha \to \alpha}\ \textsc{Var}}{\Gamma \vdash f : \mathsf{bool} \to \mathsf{bool}}\ \textsc{Inst}}{\Gamma \vdash f\ \mathsf{true} : \mathsf{bool}}\ \textsc{App}}{\Gamma \vdash (f\ 0,\ f\ \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\ \textsc{Pair}}{\varnothing \vdash \mathsf{let}\ f = \lambda z.\, z\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\ \textsc{Let}$$

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

As a counter-example, the term $\lambda f.\, (f\ 0,\ f\ \mathsf{true})$ is *ill-typed.* Indeed, as it contains no "let" construct, it is type-checked exactly as in simply-typed λ-calculus, where it is ill-typed, because f must be assigned a type τ that must simultaneously be of the form $\mathsf{int} \to \tau_1$ and $\mathsf{bool} \to \tau_2$, but there is no such type. Recall that this term is well-typed in implicitly-typed System F because f can be assigned, for instance, the polymorphic type $\forall \alpha.\, \alpha \to \alpha$.

$$\text{EML-VAR}\quad \Gamma \vdash x : \Gamma(x) \qquad \text{EML-CST}\quad \Gamma \vdash c : \Delta(c) \qquad \text{EML-ABS}\quad \frac{\Gamma, x : \tau_0 \vdash M : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, M : \tau_0 \to \tau} \qquad \text{EML-APP}\quad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1\, M_2 : \tau_1}$$

$$\text{EML-LET}\quad \frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2 : \sigma_2} \qquad \text{EML-TABS}\quad \frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma} \qquad \text{EML-TAPP}\quad \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\,\tau : [\alpha \mapsto \tau]\sigma}$$

Figure 4.4: Typing rules for $e\mathsf{ML}$ (explicitly-typed ML)

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

$$M \in e\mathsf{ML} ::= x \mid c \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid \mathsf{let}\ x : \sigma = M\ \mathsf{in}\ M \ldots$$

where τ and σ are ML-types and type schemes and not arbitrary System-F types. The typing rules for explicitly-typed terms are given on Figure 4.4.

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

Unfortunately, the converse is not true—when M is syntactically in ML and Γ and σ are well-formed in $e\mathsf{ML}$, of course. Hence, the relation $\vdash_{e\mathsf{ML}}$ cannot be defined as the restriction of $\vdash_{\mathsf{F}}$ to ML environments terms and type schemes.

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

4.6.2 Syntax-directed presentation

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

Interestingly, there is a syntax-directed presentation of implicitly-typed ML terms where the shape of typing derivations is entirely determined by the term and is thus unique. Taking the explicitly-typed view, this amounts to restricting the source terms so that there is no choice for placing type abstraction and type applications.

$$\frac{\Gamma, \vec\alpha \vdash Q : \tau}{\Gamma \vdash \Lambda\vec\alpha.Q : \forall\vec\alpha.\tau}\ \textsc{xml-Tabs} \qquad \frac{\Gamma, x : \tau_0 \vdash Q : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\,Q : \tau_0 \to \tau}\ \textsc{xml-Abs} \qquad \frac{\Gamma \vdash Q_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash Q_2 : \tau_2}{\Gamma \vdash Q_1\, Q_2 : \tau_1}\ \textsc{xml-App}$$

$$\frac{\Gamma, \vec\alpha \vdash Q_1 : \tau_1 \qquad \Gamma, x : \forall\vec\alpha.\tau_1 \vdash Q_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x : \forall\vec\alpha.\tau_1 = \Lambda\vec\alpha.Q_1\ \mathsf{in}\ Q_2 : \tau_2}\ \textsc{xml-LetGen} \qquad \frac{\forall\vec\alpha.\tau = \Gamma(x)}{\Gamma \vdash x\,\vec\tau : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{xml-VarInst} \qquad \frac{\forall\vec\alpha.\tau = \Delta(c)}{\Gamma \vdash c\,\vec\tau : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{xml-CstInst}$$

Figure 4.5: Typing rules for xML

$$\frac{\forall\vec\alpha.\tau = \Gamma(x)}{\Gamma \vdash x : \forall\vec\alpha.\tau \Rightarrow \Lambda\vec\alpha.x\,\vec\alpha}\ \textsc{norm-Var} \qquad \frac{\Gamma, \alpha \vdash M : \sigma \Rightarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}\ \textsc{norm-Tabs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}{\Gamma \vdash M\,\tau : [\alpha \mapsto \tau]\sigma \Rightarrow [\alpha \mapsto \tau]N}\ \textsc{norm-Tapp}$$

$$\frac{\forall\vec\alpha.\tau = \Delta(c)}{\Gamma \vdash c : \forall\vec\alpha.\tau \Rightarrow \Lambda\vec\alpha.c\,\vec\alpha}\ \textsc{norm-Cst}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1 \Rightarrow N_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \forall\vec\alpha.\tau \Rightarrow \Lambda\vec\alpha.Q \qquad \vec\alpha \mathrel{\#} N_1, \sigma_1}{\Gamma \vdash \mathsf{let}\ x : \sigma_1 = M_1\ \mathsf{in}\ M_2 : \forall\vec\alpha.\tau \Rightarrow \Lambda\vec\alpha.\mathsf{let}\ x : \sigma_1 = N_1\ \mathsf{in}\ Q}\ \textsc{norm-Let}$$

$$\frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \Rightarrow Q_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow Q_2}{\Gamma \vdash M_1\, M_2 : \tau_1 \Rightarrow Q_1\, Q_2}\ \textsc{norm-App} \qquad \frac{\Gamma, x : \tau_0 \vdash M : \tau \Rightarrow Q}{\Gamma \vdash \lambda x{:}\tau_0.\,M : \tau_0 \to \tau \Rightarrow \lambda x{:}\tau_0.\,Q}\ \textsc{norm-Abs}$$

Figure 4.6: Normalization of ML derivations

Let xML be the subset of explicitly-typed ML defined by the following grammar

$$\begin{array}{rcl} N \in x\mathsf{ML} & ::= & \Lambda\vec\alpha.Q \\ Q & ::= & x\,\vec\tau \mid Q\,Q \mid \lambda x{:}\tau.\,Q \mid \mathsf{let}\ x : \sigma = N\ \mathsf{in}\ Q \end{array}$$

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

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

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

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

- Strong ι-reduction, *i.e.* type β-reduction (Rule Tapp): this cancels type applications of type abstractions. As a result, elaborated terms do not contain any ι-redex.

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

Moreover, if $\Gamma \vdash M : \sigma$ holds, then $\Gamma \vdash_{x\mathsf{ML}} N : \sigma$ also holds and M and N have the same erasure. The proof is also by induction on M. The preservation of erasure is immediate. The only non obvious cases for well-typedness of N are Norm-Tapp, which performs strong ι-reduction and uses type substitution (Lemma 17), and Norm-Let, which extrudes type abstractions.

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

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

$$\textsc{xml-Tabs}\quad \frac{\Gamma, \vec{\alpha} \vdash M : \tau}{\Gamma \vdash \Lambda\vec{\alpha}.M : \forall\vec{\alpha}.\,\tau} \qquad \textsc{sml-Abs}\quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad \textsc{sml-App}\quad \frac{\Gamma \vdash a_2 : \tau_2 \qquad \Gamma \vdash a_1 : \tau_2 \to \tau_1}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{sml-LetGen}\quad \frac{\Gamma, \vec{\alpha} \vdash a_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2} \qquad \textsc{sml-VarInst}\quad \frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash x : [\vec{\alpha} \mapsto \vec{\tau}]\tau} \qquad \textsc{sml-CstInst}\quad \frac{\forall\vec{\alpha}.\,\tau = \Delta(c)}{\Gamma \vdash c : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

Then, the judgments $\Gamma \vdash_{\mathsf{ML}} a : \tau$ and $\Gamma \vdash_{s\mathsf{ML}} a : \tau$ are equivalent.

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

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

Lemma 29 (Type Substitution) *Typings are stable by substitution.*
If $\Gamma \vdash a : \tau$ *then* $\varphi\Gamma \vdash a : \varphi\tau$*. for any substitution* φ*.*

$$\text{ML-ABS}\quad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad \text{ML-APP}\quad \frac{\Gamma \vdash a_2 : \tau_2 \qquad \Gamma \vdash a_1 : \tau_2 \to \tau_1}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\text{ML-LETGEN}\quad \frac{\Gamma \vdash a_1 : \tau_1 \qquad \vec{\alpha} \# \Gamma \qquad \Gamma, x : \forall \vec{\alpha}.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2} \qquad \text{ML-VARINST}\quad \frac{\forall \vec{\alpha}.\, \tau = \Gamma(x)}{\Gamma \vdash x : [\vec{\alpha} \mapsto \vec{\tau}]\tau} \qquad \text{ML-VARCST}\quad \frac{\forall \vec{\alpha}.\, \tau = \Delta(c)}{\Gamma \vdash c : [\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

Figure 4.7: Syntax-directed rules for ML

4.6.3 Type soundness for ML

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

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

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

To establish subject-reduction in ML, let a_1 be an implicitly-typed ML term such that both $\vec{\alpha} \vdash_{\mathsf{ML}} a_1 : \sigma$ and $a_1 \longrightarrow a_2$ hold. There exists an explicitly-typed term M_1 such that $\vec{\alpha} \vdash_{e\mathsf{ML}} M_1 : \sigma$ and $\lceil M_1 \rceil = a_1$. By normalization, we may elaborate M_1 into a term N_1 of $x\mathsf{ML}$ such that $\vec{\alpha} \vdash_{x\mathsf{ML}} N_1 : \sigma$ and the $\lceil N_1 \rceil = \lceil M_1 \rceil$. Moreover, N_1 is by construction ι-normal. Since $x\mathsf{ML}$ is a subset of System F, we have $\vec{\alpha} \vdash_{\mathsf{F}} N_1 : \sigma$. By inverse simulation in System F (Lemma 22), there exists N_2 in F whose type erasure is a_2 and such that $N_1 \longrightarrow_\beta N_2$ (since N_1 is ι-normal). We show below that there exists a strong ι-reduction M_2 of N_2 that is in $x\mathsf{ML}$ and such that $\vec{\alpha} \vdash_{x\mathsf{ML}} N_2 : \sigma$. Therefore, we have $\vec{\alpha} \vdash_{e\mathsf{ML}} M_2 : \sigma$ and since the type erasure of M_2 is that of N_2, *i.e.* a_2, we have $\vec{\alpha} \vdash_{\mathsf{ML}} a_2 : \sigma$, as expected.

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

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

$$(\lambda x\!:\!\tau.\,Q)\;V \longrightarrow [x \mapsto V]Q$$

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

$$\mathsf{let}\; x : \forall\vec{\alpha}.\,\tau = V \;\mathsf{in}\; Q \longrightarrow [x \mapsto V]Q$$

then occurrences of x in Q, which are of the form $x\;\vec{\tau}$, become $V\;\vec{\tau}$ and may contain ι-redexes—which are not allowed in $x\mathsf{ML}$. Fortunately, V is necessarily of the form $\Lambda\vec{\alpha}.V'$ where the arity of $\vec{\alpha}$ is equal to that of $\vec{\tau}$. Hence, we may immediately perform a sequence of ι-reduction that brings the term back into $x\mathsf{ML}$ and in ι-normal form. Notice however that this ι-redex is not in general in a call-by-value evaluation context. Indeed, x may appear under an abstraction in M. Hence, this is a strong reduction step.

For type reduction, we need to ensure that strong ι-reduction is also type-preserving. This is an easy auxiliary proof—but not a consequence of subject reduction, which we have only proved for reduction in call-by-value evaluation contexts.

4.7 Ommitted proofs and answers to exercises

Solution of Exercise 24

As in the case where pairs are primitive, we introduce one constructor $(\cdot,\cdot)$ of arity 2 and and two destructors proj_1 and proj_2 of arity 1, with the following types in Δ

$$\begin{array}{rl} Pair: & \forall\alpha_1.\,\forall\alpha_2.\,\alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ proj_i: & \forall\alpha_1.\,\forall\alpha_2.\,\alpha_1 \times \alpha_2 \to \alpha_i \end{array}$$

and the two reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (Pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \tag{δ_i}$$

We then only need to verify that δ_i preserves types and ensure progress.

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

Case Progress: Assume that $\Gamma \vdash M : \tau$ and M is of the form $proj_i\ \tau_1\ \tau_2\ V$. By the inversion lemma, τ must be a product type $\tau_1 \times \tau_2$ such that $\Gamma \vdash V : \tau_1 \times \tau_2$. By the classification lemma, V must be a pair, *i.e.* of a form $Pair\ \tau_1\ \tau_2\ V_1\ V_2$. Hence, M reduces to V_i by δ_i. ∎

Solution of Exercise 25

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

$$\mathsf{ifcase}\ \tau\ \mathsf{true}\ V_1\ V_2 \longrightarrow V_1 \qquad\qquad \mathsf{ifcase}\ \tau\ \mathsf{false}\ V_1\ V_2 \longrightarrow V_2$$

This extension is sound.

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

An easy fix is to introduce $\mathsf{iflazy}\ \tau\ M_0\ M_1\ M_2$ as syntactic sugar for

$$(\mathsf{ifcase}\ \ \tau\ M_0\ (\lambda()\!:\!\mathsf{unit}.\,M_1)\ (\lambda()\!:\!\mathsf{unit}.\,M_2))\ ()$$

and exposing it to the user, while hiding the primitive ifcase from the user. ∎

Solution of Exercise 26

1) We introduce a new unary type constructor list; two constructors $\mathsf{Nil}\cdot$ and Cons of types $\forall\alpha.\,\mathsf{list}\ \alpha$ and $\forall\alpha.\,\alpha \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$; and one destructor $\mathsf{matchlist}\cdot\cdot\cdot$ of type:

$$\forall\alpha\beta.\,\mathsf{list}\ \alpha \to \beta \to (\alpha \to \mathsf{list}\ \alpha \to \beta) \to \beta$$

with the two reduction rules:

$$\begin{array}{l}\mathsf{matchlist}\ \tau\ (\mathsf{Nil}\ \tau')\ V_n\ V_c \longrightarrow V_n\\ \mathsf{matchlist}\ \tau\ (\mathsf{Cons}\ \tau'\ V_h\ V_t)\ V_n\ V_c \longrightarrow V_c\ V_h\ V_t\end{array}$$

2) Ommitted. ■

Solution of Exercise 27

In ML, we may define the datatype:

```
type any = Fold of (any → any)
```

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

$$\mathsf{unfold}\ (\mathsf{Fold}\ V) \longrightarrow V \qquad \delta_{any}$$

Let us check soundness of this extension:

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

Case Progress: Assume that $\Gamma \vdash \mathsf{unfold}\ V : \tau$. By inversion, τ must be $\mathsf{any} \to \mathsf{any}$ and $\Gamma \vdash V : \mathsf{any}$ holds. By classification, V must be $\mathsf{Fold}\ V_0$. Hence, $\mathsf{unfold}\ V$ reduces.

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

```
let zfix g = (fun x → x x) (fun z → g (fun v → z z v))
```

We may implement zfix in ML without recursive tyes as:

```
type any = Fold of (any → any);;
let unfold (Fold x) = x;;
let zfix g =
  (fun x → unfold (x (Fold x)))
    (fun z → Fold (g (fun v → unfold ((unfold z) z) v)));;
```

■

Proof of Lemma 16

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

Case M is x: By (1), it must be the case that τ is equal to τ_0. Hence, the goal (3) is $\Gamma, \Gamma' \vdash M_0 : \tau_0$, which follows from the hypothesis (2) by weakening.

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

Case M is c: By (1), $c : \tau$ is in Δ. Hence, the goal (3) follows by Rule Var.

Case M is $\lambda y{:}\tau_1.\, M_1$: By (1), τ is of the form $\tau_2 \to \tau_1$ and $\Gamma, x : \tau_0, \Gamma', y : \tau_2 \vdash M_1 : \tau_1$ holds. By induction hypothesis, we have $\Gamma, \Gamma', y : \tau_2 \vdash [x \mapsto M_0]M_1 : \tau_1$. By rule Abs, we have $\Gamma, \Gamma' \vdash \lambda y{:}\tau_2.\, [x \mapsto M_0]M_1 : \tau_1$, which is the goal (3).

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

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

Proof of Lemma 17

The proof is by induction on M using inversion of the typing derivation of $\Gamma, \alpha, \Gamma' \vdash M : \tau$ (**1**). We write θ for $[\alpha \mapsto \tau]$. We must show $\Gamma, \theta\Gamma' \vdash \theta M : \theta\tau$ (**2**).

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

Case M is c: By (1), we have $c : \tau$ is in Δ and τ is closed. Hence $\theta\tau$ is equal to τ and $c : \theta\tau$ is still in Δ. Thus, the conclusion (2) follows by Rule Const.

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

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

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

Solution of Exercise 30

Take, for instance, $\lambda f.\, \lambda x.\, \lambda y.\, (f\ y,\ f\ x)$ for a_1 (notice the inverse order of fields in the pair) and $\lambda f.\, \lambda x.\, \lambda y.\, (f\ (f\ x),\ f\ (f\ y))$ for a_2. ∎

Solution of Exercise 31

Choose, for instance,

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

for τ_0. We recover τ_1 by choosing the constant functions $\lambda\alpha.\, \alpha_i$ for φ_i and τ_2 by choosing the identity $\lambda\alpha.\, \alpha$ for both φ_1 and φ_2. ∎

Solution of Exercise 32

1) Both directions follow from rule Inst-Gen, just applying the substitution $\alpha \mapsto \alpha$ for the direct implication and just generalizing over α for the reverse.

2) Rule Distrib-Right is a particular case of Distributivity, indeed. Assuming $\alpha \notin \mathsf{ftv}(\tau_1)$, and using the previous equivalence (**1**), we have

$$\dfrac{\dfrac{}{\forall\alpha.(\tau_1 \to \tau_2) \le (\forall\alpha.\tau_1) \to (\forall\alpha.\tau_2)}\,\textsc{Distributivity} \qquad \dfrac{\tau_1 \le \forall\alpha.\tau_1 \ (1)}{(\forall\alpha.\tau_1) \to (\forall\alpha.\tau_2) \le \tau_1 \to (\forall\alpha.\tau_2)}\,\textsc{Congruence-}\to}{\forall\alpha.(\tau_1 \to \tau_2) \le \tau_1 \to (\forall\alpha.\tau_2)}\,\textsc{Trans}$$

Conversely, we have the following derivation:

$$\textsc{Distrib-Right}\ \dfrac{\textsc{Congruence-}\forall\ \dfrac{\textsc{Congruence-}\to\ \dfrac{\textsc{Inst-Gen}\ \forall\alpha.\tau_1 \le \tau_1}{\tau_1 \to \tau_2 \le (\forall\alpha.\tau_1) \to \tau_2}}{\forall\alpha.(\tau_1 \to \tau_2) \le \forall\alpha.(\forall\alpha.\tau_1) \to \tau_2}}{\forall\alpha.(\tau_1 \to \tau_2) \le (\forall\alpha.\tau_1) \to (\forall\alpha.\tau_2)}$$

■

Solution of Exercise 33

We extend the λ-calculus with a binary constructor *Pair* and two unary destructors $proj_i$ for i in $\{1, 2\}$ with the δ-rules:

$$proj_i\ (Pair\ v_1\ v_2) \longrightarrow_\delta v_i$$

The reduction $proj_1\ (Pair\ v\ (\lambda x. Pair\ Pair)) \longrightarrow_\delta v$ is correct, even though the right component of the pair is ill-typed, hence δ-reduction is larger than the type-erasure of δ-reduction on explicitly typed terms. Still, it contains it (direct simulation); and it does not contains more (inverse simulation) when we restrict to well-typed expressions. Both cases are really easy:

<u>Proof</u>: Let M be of the form $proj_i\ \tau_1\ \tau_2\ V_0$ such that $\Gamma \vdash M : \tau$. By inversion of typing rules, $\Gamma \vdash V_0 : \tau_1 \times \tau_2$. By the classification lemma, V_0 is of the form $Pair\ \tau_1\ \tau_2\ V_1\ V_2$. Observe that M reduces to V_i (**1**); and $\lceil M \rceil$ is $proj_i\ (pair\ \lceil V_1 \rceil\ \lceil V_2 \rceil)$ which reduces to $\lceil V_i \rceil$ (**2**).

Case direct: Assume $M \longrightarrow_\delta M'$. Then, M' is V_i and by (2), $\lceil M \rceil \longrightarrow_\delta \lceil V_i \rceil$.

Case inverse: Assume that $\lceil M \rceil \longrightarrow_\delta a$. Since reduction is deterministic in the untyped calculus a must be $\lceil V_i \rceil$. Hence, we may take V_i for M'.

■

Proof of Lemma 21

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

Case $(\Lambda(\alpha)\ M_0)\ \tau \longrightarrow_\iota M_0[\tau/\alpha]$: Observe that both M and M' erases to $\lceil M_0 \rceil$.

Case $(\lambda x.\, M_1)\ M_2 \longrightarrow_\beta M_1[M_2/x]$: Then M erases to $(\lambda x.\lceil M_1 \rceil)\ \lceil M_1 \rceil$ which reduces to $\lceil M_1 \rceil[\lceil M_2 \rceil/x]$ which is the erasure of M'

Case $\mathsf{proj}_i\ \tau_1\ \tau_2\ (V_1, V_2) \longrightarrow_\delta V_i$: The conclusion follows by assumption on δ-rules.

Case M is $E[N]$ and M' is $E[N']$ and $N \longrightarrow_z N'$: By induction hypothesis, we know that a certain relation (equality when z is ι or $\longrightarrow_z$ otherwise) holds between $\lceil M \rceil$ and $\lceil N \rceil$. By rule congruence for ι and rule Context otherwise, the same relation holds between $\lceil E[M] \rceil$ and $\lceil E \rceil[\lceil N \rceil]$, *i.e.* between $\lceil M \rceil$ and $\lceil N \rceil$.

Proof of Lemma 25

Case v is $\lambda x.\, a_1$: By inversion of type erasure, M is of the form $\mathcal{R}[\lambda x{:}\tau.\, M_1]$ where $\lceil M_1 \rceil$ is a_1. Since $\mathcal{R}$ is ι-normal, it is of the form $\Lambda\overline{\alpha}.[]\ \overline{\tau}$. since $\lambda x{:}\tau.\, M_1$ is an arrow type, $\overline{\tau}$ must be empty.

Case v is a partial application $c\ v_1\ \ldots v_n$: We show that then V is $\mathcal{R}[c\ \vec{\tau}\ V_1\ \ldots V_n]$ with $\lceil V_i \rceil = v_i$ by induction on n. If n is zero, then by inversion of type erasure, M is of the form $\mathcal{R}[c]$ as expected. Otherwise, by inversion of type erasure, M is an application $\mathcal{R}_n[M_1\ M_2]$ where $\lceil M_1 \rceil$ is the partial application $c\ v_1\ \ldots v_{n-1}$ and $\lceil M_2 \rceil$ is v_n. By induction hypothesis M_1 is $\mathcal{R}_1[c\ \vec{\tau}\ V_1\ \ldots V_{n-1}]$ with $\lceil V_i \rceil = v_i$. Since $\mathcal{R}_1$ is in an evaluation context, it is ι-normal, hence of the form $\Lambda\bar{\alpha}_1.[]\ \bar{\tau}_1$. From the arity of c, the type of M_1 is an arrow type. Thus $\bar{\tau}_1$ must be empty. Since $\mathcal{R}$ is applied to M_2 it cannot be a type abstraction either. Hence, $\mathcal{R}_1$ is empty. Moreover, by induction hypothesis M_2 is a value V_n. Hence M is $\mathcal{R}_n[c\ \vec{\tau}\ V_1\ \ldots V_{n-1}\ V_n]$, as expected.

Proof of Corollary 26

By Lemma 24, M is of the form $\mathcal{R}[M_0\ M_2]$ where $\lceil M_0 \rceil$ is a value v, which is either $\lambda x.\, a_1$ or the partial application $c\ v_1\ \ldots v_{n-1}$ and $\lceil M_2 \rceil$ is v. Since $\mathcal{R}$ is an evaluation context, $M_0\ M_2$ is in ι-normal form. Since $[]\ M_2$ is an evaluation context, M_0 is in ι-normal form. By Lemma 25, M_0 a value V_0. Since $V_0\ []$ is an evaluation context, M_2 is in ι-normal form. By 25, it must be a value V_0.

Moreover, by Lemma 25, V_0 is either

- $\Lambda\overline{\alpha}.\lambda x{:}\tau.\, M_1$. Since V_0 is in application position $\overline{\alpha}$ must actually be empty. Then M is of the form $\mathcal{R}[(\lambda x{:}\tau.\, M_1)\ V]$, as expected.

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

Solution of Exercise 34

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

Chapter 5

Type reconstruction

5.1 Introduction

We have viewed a type system as a 3-place *predicate* over a type environment, a term, and a type. So far, we have been concerned with *logical* properties of the type system, namely subject reduction and progress. However, one should also study its *algorithmic* properties: is it decidable whether a term is well-typed?

We have seen three different type systems, simply-typed λ-calculus, ML, and System F, of increasing expressiveness. In each case, we have presented an explicitly-typed and an implicitly-typed version of the language and shown a close correspondence between the two views, thanks to a type-erasing semantics.

We argued that the explicitly-typed version is often more convenient for studying the meta-theoretical properties of the language. Which one should we used for checking well-typedness? That is, in which language should we write programs?

The typing judgment is *inductively defined*, so that, in order to prove that a particular instance holds, one exhibits a *type derivation*. A type derivation is essentially a version of the program where *every* node is annotated with a type. *Checking* that a type derivation is correct is usually easy: it basically amounts to checking equalities between types. However, type derivations are too verbose to be tractable by humans! Requiring every node to be type-annotated is not practical.

A more practical, common approach consists in requesting just enough annotations to allow types to be reconstructed in a *bottom-up* manner. In other words, one seeks an *algorithmic reading* of the typing rules, where, in a judgment $\Gamma \vdash M : \tau$, the parameters Γ and M are *inputs*, while the parameter τ is an *output*. Moreover, typing rules should be such that a type appearing as output in a conclusion should also appear as output in a premise or as input in the conclusion; and input in the premises should be input of the conclusion or an output of other premises.

This way, types need never be guessed, just looked up into the typing context, instanti-

ated, or checked for equality. This is exactly the situation with explicitly-typed presentations of the typing rules. This is also the traditional approach of Pascal, C, C++, Java, *etc.*: formal procedure parameters, as well as local variables, are assigned explicit types. The types of expressions are synthesized bottom-up.

However, this implies a lot of redundancies: Parameters of *all* functions need to be annotated, even when their types are obvious from context; Primitive let-bindings, recursive definitions, injection into sum types need to be annotated. As the language grows, more and more constructs require type annotations, *e.g.* type applications and type abstractions. Type annotations may quickly obfuscate the code and large explicitly-typed terms are so verbose that they become intractable by humans! Hence, programming in the implicitly-typed version is more appealing.

For simply-typed λ-calculus and ML, it turns out that this is possible: *whether a term is well-typed is decidable*, even when no type annotations are provided! We first present type inference in the case of simply-typed λ-calculus taking advantage of the simplicity to introduce type constraints as a useful intermediate to mediate between the typing rules and the type-inference algorithms. We then extend type-constraint to perform type inference for ML.

For System F, type inference is undecidable. Since programming in explicitly-typed System F is not practically feasible, some amount of type reconstruction must still be done. Typically, the algorithm is incomplete, *i.e.* it rejects terms that are perhaps well-typed, but the user may always provide more annotations—and at least the fully annotated version is always accepted if well-typed. We will present very briefly several techniques for type reconstruction in System F.

5.2 Type inference for simply-typed λ-calculus

The type inference algorithm for simply-typed λ-calculus, is due to Hindley. The idea behind the algorithm is simple. Because simply-typed λ-calculus is a *syntax-directed* type system, an unannotated term determines an isomorphic *candidate type derivation*, where all types are unknown: they are distinct *type variables.* For a candidate type derivation to become an actual, valid type derivation, every type variable must be instantiated with a type, subject to certain *equality constraints* on types. For instance, at an application node, the type of the operator must match the domain type of the operator.

Thus, type inference for the simply-typed λ-calculus decomposes into *constraint generation* followed by *constraint solving.* Simple types are *first-order terms.* Thus, solving a collection of equations between simple types is *first-order unification.* First-order unification can be performed incrementally in quasi-linear time, and admits particularly simple *solved forms.*

$$
\begin{array}{rcll}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle & = & \Gamma(x) = \tau & \\
\langle\!\langle \Gamma \vdash \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\langle\!\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle\!\rangle \wedge \tau = \alpha_1 \to \alpha_2) & \text{if } \alpha_1, \alpha_2 \# \Gamma, \tau \\
\langle\!\langle \Gamma \vdash a_1\, a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle \Gamma \vdash a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_2 : \alpha \rangle\!\rangle) & \text{if } \alpha \# \Gamma, \tau
\end{array}
$$

Figure 5.1: constraint generation for simply-typed λ-calculus

5.2.1 Constraints

At the interface between the constraint generation and constraint solving phases is the *constraint language*. It is a *logic:* a *syntax*, equipped with an *interpretation* in a model.

There are two syntactic categories: *types* and *constraints*.

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid F\,\vec{\tau} \\
C & ::= & \mathsf{true} \mid \mathsf{false} \mid \tau = \tau \mid C \wedge C \mid \exists \alpha.C
\end{array}
$$

A type is either a *type variable* α or an arity-consistent application of a *type constructor* F. (The type constructors are unit, $\times$, $+$, $\to$, etc.) An atomic constraint is truth, falsity, or an *equation* between types. Compound constraints are built on top of atomic constraints via *conjunction* and *existential quantification* over type variables.

Constraints are interpreted in the Herbrand universe, that is, in the set of *ground types:*

$$\mathbf{t} \quad ::= \quad F\,\vec{\mathbf{t}}$$

Ground types contain no variables. The base case in this definition is when F has arity zero. We assume that there should be at least one constructor of arity zero, so that the model is non-empty. A *ground assignment* ϕ is a total mapping of type variables to ground types. By homomorphism, a ground assignment determines a total mapping of types to ground types.

The interpretation of constraints takes the form of a judgment, $\phi \vdash C$, pronounced: ϕ *satisfies* C, or ϕ *is a solution of* C. This judgment is inductively defined:

$$
\phi \vdash \mathsf{true} \qquad \frac{\phi\tau_1 = \phi\tau_2}{\phi \vdash \tau_1 = \tau_2} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi[\alpha \mapsto \mathbf{t}] \vdash C}{\phi \vdash \exists \alpha.C}
$$

A constraint C is *satisfiable* if and only if there exists a ground assignment ϕ that satisfies C. We write $C_1 \equiv C_2$ when C_1 and C_2 have the same solutions. The problem *"given a constraint C, is C satisfiable?"* is *first-order unification.*

Type inference is reduced to constraint solving by defining a mapping $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ of *candidate judgments* to constraints, as given in Figure 5.1. Thanks to the use of existential quantification, the names that occur free in $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ are a subset of those that occur free in Γ or τ. This allows the freshness side conditions to remain *local*—there is no need to informally require "globally fresh" type variables.

5.2.2 A detailed example

Let us perform type inference for the closed term $\lambda f x y.\,(f\ x,\ f\ y)$. The problem is to *construct* and *solve* the constraint $\langle\!\langle \varnothing \vdash \lambda f x y.\,(f\ x,\ f\ y) : \alpha_0 \rangle\!\rangle$, say C. It is possible (and, for a human, easier) to mix these tasks. A machine, however, could generate and solve in two successive phases. There are several advantages in doing this. This leads to simpler, easier to maintain code, as the generation of constraints deals with the complexity of the source language which solving may ignore; moreover, adding new construct to the language does not (in general) require new forms of constraints and can thus reuse the solving algorithm unchanged.

Solving the constraint means to find all possible ground assignments for α_0 that satisfy the constraint. Typically, this is done by transforming the constraint into successive equivalent constraints until some constraint that is obviously satisfiable and from which solutions may be directly read.

Performing constraint generation for the 3 λ-abstractions, we have:

$$C \;=\; \exists \alpha_1\alpha_2. \left(\begin{array}{l} \exists \alpha_3\alpha_4. \left(\begin{array}{l} \exists \alpha_5\alpha_6. \left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x,\ f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_4 = \alpha_5 \to \alpha_6 \end{array} \right) \\ \alpha_2 = \alpha_3 \to \alpha_4 \end{array} \right) \\ \alpha_0 = \alpha_1 \to \alpha_2 \end{array} \right)$$

In the following, let Γ stand for $(f : \alpha_1; x : \alpha_3; y : \alpha_5)$. We may hoist up existential quantifiers, using the rule:

$$\boxed{(\exists \alpha.C_1) \wedge C_2 \;\;\equiv\;\; \exists \alpha.(C_1 \wedge C_2)} \qquad \text{if } \alpha \# C_2$$

Hence, hoisting α_3 and α_4, and α_5 and α_6 twice, we get:

$$C \equiv \exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6. \left(\begin{array}{l} \langle\!\langle \Gamma \vdash (f\ x,\ f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_4 = \alpha_5 \to \alpha_6 \;\;\wedge\;\; \alpha_2 = \alpha_3 \to \alpha_4 \;\;\wedge\;\; \alpha_0 = \alpha_1 \to \alpha_2 \end{array} \right)$$

We may eliminate a type variable that has a defining equation with the rule:

$$\boxed{\exists \alpha.(C \wedge \alpha = \tau) \;\;\equiv\;\; [\alpha \mapsto \tau]C} \qquad \text{if } \alpha \# \tau$$

By successive elimination of α_4 then α_2, we get:

$$C \;\equiv\; \exists \alpha_1\alpha_3\alpha_5\alpha_6. \left(\begin{array}{l} \langle\!\langle \Gamma \vdash (f\ x,\ f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 \end{array} \right)$$

Let us now perform constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable (α_6).

$$C \;\equiv\; \exists \left\{ \begin{array}{l} \alpha_1\alpha_3\alpha_5 \\ \alpha_6\alpha_7\alpha_8 \end{array} \right\}. \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_7 \times \alpha_8 = \alpha_6 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0 \end{array} \right) \;\equiv\; \exists \left\{ \begin{array}{l} \alpha_1\alpha_3\alpha_5 \\ \alpha_7\alpha_8 \end{array} \right\}. \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \\ \quad \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array} \right)$$

Let us focus on the first application, perform constraint generation for the variables f and x (recall that Γ stands for $(f : \alpha_1; x : \alpha_3; y : \alpha_5)$), and eliminate a type variable (α_9):

$$C_1 = \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle = \exists \alpha_9. \begin{pmatrix} \langle\!\langle \Gamma \vdash f : \alpha_9 \to \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash x : \alpha_9 \rangle\!\rangle \end{pmatrix} = \exists \alpha_9. \begin{pmatrix} \alpha_1 = \alpha_9 \to \alpha_7 \\ \alpha_3 = \alpha_9 \end{pmatrix} \equiv \alpha_1 = \alpha_3 \to \alpha_7 = C_2$$

Applying this simplification under a context, with the rule:

$$\boxed{C_1 \equiv C_2 \;\Rightarrow\; \mathcal{R}[C_1] \equiv \mathcal{R}[C_2]}$$

we have:

$$C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 \end{pmatrix}$$

We may simplify the right-hand application analogously.

$$C \equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \;\wedge\; \alpha_1 = \alpha_5 \to \alpha_8 \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 \end{pmatrix}$$

We may apply transitivity at α_1, structural decomposition, and eliminate three type variables (α_1, α_5, α_8):

$$\begin{aligned} C &\equiv \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \;\wedge\; \alpha_3 = \alpha_5 \;\wedge\; \alpha_7 = \alpha_8 \\ \alpha_0 = \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 \end{pmatrix} \\ &\equiv \exists \alpha_3 \alpha_7. \big(\alpha_0 = (\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 z \big) \end{aligned}$$

We have now reached a *solved form*. To sum up, we have checked the following equivalence holds:

$$\langle\!\langle \varnothing \vdash \lambda f x y. (f\ x,\ f\ y) : \alpha_0 \rangle\!\rangle \equiv \exists \alpha_3 \alpha_7. \big((\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_0 \big)$$

Hence, the ground types of $\lambda f x y. (f\ x,\ f\ y)$ are all ground types of the form

$$(\mathbf{t}_3 \to \mathbf{t}_7) \to \mathbf{t}_3 \to \mathbf{t}_3 \to \mathbf{t}_7 \times \mathbf{t}_7$$

In other words, $(\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7$ is a *principal type* for $\lambda f x y. (f\ x,\ f\ y)$.

The language OCaml implements a form of this type inference algorithm:

```
# fun f x y → (f x, f y);;
- : ('a → 'b) → 'a → 'a → 'b * 'b = ⟨fun⟩
```

This technique is used also by Standard ML and Haskell.

In the simply-typed λ-calculus, type inference works just as well for *open* terms. For instance, the term $\lambda x y. (f\ x,\ f\ y)$ has a free variable, namely f. The type inference problem is to *construct* and *solve* the constraint $\langle\!\langle f : \alpha_1 \vdash \lambda x y. (f\ x,\ f\ y) : \alpha_2 \rangle\!\rangle$. We have already done so... with only a slight difference: α_1 and α_2 are now free, so they cannot be eliminated.

One can check the following equivalence:

$$\langle\!\langle f : \alpha_1 \vdash \lambda xy.(f\ x,\ f\ y) : \alpha_2 \rangle\!\rangle \ \equiv\ \exists \alpha_3\alpha_7.\big(\ \alpha_1 = \alpha_3 \to \alpha_7 \ \ \wedge\ \ \alpha_2 = \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7\ \big)$$

In other words, the ground *typings* of $\lambda xy.(f\ x,\ f\ y)$ are all ground typings of the form:

$$((f : \mathbf{t}_3 \to \mathbf{t}_7),\ \ \mathbf{t}_3 \to \mathbf{t}_3 \to \mathbf{t}_7 \times \mathbf{t}_7)$$

Remember that a typing is a pair of an environment and a type.

5.2.3 Soundness and completeness of type inference

Definition 2 (Typing) *A pair* (Γ, τ) *is a* typing *of* a *if and only if* $\mathsf{dom}(\Gamma) = \mathsf{fv}(a)$ *and the judgment* $\Gamma \vdash a : \tau$ *is valid.*

The type inference problem is to determine whether a term a admits a typing, and, if possible, to exhibit a description of the set of all of its typings.

Up to a change of universes, the problem reduces to finding the *ground typings* of a term. (For every type variable, introduce a nullary type constructor. Then, ground typings in the extended universe are in one-to-one correspondence with typings in the original universe.)

Theorem 15 (Soundness and completeness) $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

In other words, assuming $\mathsf{dom}(\Gamma) = \mathsf{fv}(a)$, ϕ satisfies the constraint $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ if and only if $(\phi\Gamma, \phi\tau)$ is a (ground) typing of a. The direct implication is soundness; the reverse implication is completeness. The proof is by structural induction over a. (Proof p. 124)

Exercise 35 (Recommended) *Write the details of the proof.* □

Corollary 30 *Let* $\mathsf{fv}(a) = \{x_1, \ldots, x_n\}$*, where* $n \geq 0$*. Let* $\alpha_0, \ldots, \alpha_n$ *be pairwise distinct type variables. Then, the ground typings of* a *are described by* $((x_i : \phi\alpha_i)_{i\in 1..n},\ \phi\alpha_0)$ *where* ϕ *ranges over all solutions of* $\langle\!\langle (x_i : \alpha_i)_{i\in 1..n} \vdash a : \alpha_0 \rangle\!\rangle$.

Corollary 31 *Let* $\mathsf{fv}(a) = \varnothing$*. Then,* a *is well-typed if and only if* $\exists \alpha.\langle\!\langle \varnothing \vdash a : \alpha \rangle\!\rangle \equiv \mathsf{true}$.

5.2.4 Constraint solving

A constraint solving algorithm is typically presented as a (non-deterministic) system of *constraint rewriting rules* that must enjoy the following properties: reduction is meaning-preserving, *i.e.* $C_1 \longrightarrow C_2$ implies $C_1 \equiv C_2$; reduction is terminating; and every normal form is either “false” (literally) or satisfiable. The normal forms are called *solved forms*.

Our constraints are equations on first-order terms. They can be solved by first-order unification. The algorithm can be described as constraint solving. However, in order to

$$
\begin{array}{rcll}
(\exists\bar{\alpha}.U_1) \wedge U_2 & \longrightarrow & \exists\bar{\alpha}.(U_1 \wedge U_2) & \text{(extrusion)} \\
& & \text{if } \bar{\alpha} \mathrel{\#} U_2 & \\
\alpha = \epsilon \wedge \alpha = \epsilon' & \longrightarrow & \alpha = \epsilon = \epsilon' & \text{(fusion)} \\
F\,\vec{\alpha} = F\,\vec{\tau} = \epsilon & \longrightarrow & \vec{\alpha} = \vec{\tau} \wedge F\,\vec{\alpha} = \epsilon & \text{(decomposition)} \\
F\,\tau_1\,\ldots\,\tau_i\,\ldots\,\tau_n = \epsilon & \longrightarrow & \exists\alpha.(\alpha = \tau_i \wedge F\,\tau_1\,\ldots\,\alpha\,\ldots\,\tau_n = \epsilon) & \text{(naming)} \\
& & \text{if } \tau_i \text{ is not a variable} \wedge \alpha \mathrel{\#} \tau_1, \ldots, \tau_n, \epsilon & \\
F\,\vec{\tau} = F'\,\vec{\tau}' = \epsilon & \longrightarrow & \mathsf{false} & \text{(clash)} \\
& & \text{if } F \neq F' & \\
U & \longrightarrow & \mathsf{false} & \text{(occurs check)} \\
& & \text{if } U \text{ is cyclic} & \\
\mathcal{U}[\mathsf{false}] & \longrightarrow & \mathsf{false} & \text{(error propag.)} \\
\alpha = \alpha = \epsilon & \longrightarrow & \alpha = \epsilon & \text{(elim dupl.)} \\
F\,\vec{\tau} & \longrightarrow & \mathsf{true} & \text{(elim triv.)} \\
U \wedge \mathsf{true} & \longrightarrow & U & \text{(elim true)}
\end{array}
$$

Figure 5.2: Solving unification constraints

describe an efficient algorithm, we first extend the syntax of constraints and replace ordinary binary equations with *multi-equations*, following Pottier and Rémy (2005, §10.6):

$$U ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid U \wedge U \mid \exists\bar{\alpha}.U$$

A multi-equation ϵ is a multi-set of types. Its interpretation is given by

$$\frac{\forall\tau \in \epsilon, \quad \phi\tau = \mathbf{t}}{\phi \vdash \epsilon}$$

That is, ϕ satisfies ϵ if and only if ϕ maps all members of ϵ to a single ground type.

Simplification rules are given in Figure 5.2. (See Pottier and Rémy (2005, §10.6) for a detailed presentation.) The last three rules in gray are administrative.

The occurs check is defined as follows: we say that α *dominates* β (with respect to U) if U contains a multi-equation of the form $F\,\tau_1\,\ldots\beta\,\ldots\,\tau_n = \alpha = \ldots$. A constraint U is *cyclic* if and only if its domination relation is cyclic. A cyclic constraint is unsatisfiable: indeed, if ϕ satisfies U and if α is a member of a cycle, then the ground type $\phi\alpha$ must be a strict subterm of itself, a contradiction. Thus, the occurs-check rewriting rule is meaning-preserving.

A *solved form* is either false or $\exists\bar{\alpha}.U$, where U is a conjunction of multi-equations, every multi-equation contains at most one non-variable term, no two multi-equations share a variable, and the domination relation is acyclic. Every solved form that is not false is satisfiable. Indeed, a solution is easily constructed by well-founded recursion over the domination relation.

Remarks Viewing a unification algorithm as a system of rewriting rules makes it easy to explain and reason about.

In practice, following Huet (1976), first-order unification is implemented on top of an efficient *union-find* data structure (Tarjan, 1975). Its time complexity is quasi-linear (*i.e.* growing in the inverse of the Ackermann function).

Unification on first-order terms can also be implemented in linear time, but with a more complex algorithm and a higher constant that makes it behave worse than the quasi-linear time algorithm. Moreover, while the quasi-linear time algorithm works as well when types are regular trees— by just removing the occur check—the linear time algorithm only works with finite trees and thus cannot be used for type inference in the presence of equi-recursive types.

Closing remarks Thanks to type inference, *conciseness* and *static safety* are not incompatible. Furthermore, an inferred type is sometimes *more general* than a programmer-intended type. Type inference helps reveal unexpected generality.

5.3 Type inference for ML

Two presentations of type inference for Damas and Milner's type system are possible: One of Milner's classic algorithms 1978, $\mathcal{W}$ or $\mathcal{J}$; see Pottier's old course notes for details (Pottier, 2002, §3.3); or a constraint-based presentation Pottier and Rémy (2005). We favor the latter, but quickly review the former first.

5.3.1 Milner's Algorithm $\mathcal{J}$

Milner's Algorithm $\mathcal{J}$ expects a pair $\Gamma \vdash a$, produces a type τ, and uses two global variables, $\mathcal{V}$ and φ. Variable $\mathcal{V}$ is an infinite *fresh supply* of type variables; φ is an *idempotent substitution* (of types for type variables), initially the identity. The fresh primitive is defined as:

$$\mathsf{fresh} \;=\; \mathsf{do}\; \alpha \in \mathcal{V}; \quad \mathsf{do}\; \mathcal{V} \leftarrow \mathcal{V} \setminus \{\alpha\}; \quad \mathsf{return}\; \alpha$$

The Algorithm $\mathcal{J}$ is given on Figure 5.3 in monadic style. The algorithm mixes *generation* and *solving* of equations. This lack of modularity leads to several weaknesses: proofs are more difficult; correctness and efficiency concerns are not clearly separated (if implemented literally, the algorithm is exponential in practice); adding new language constructs duplicates solving of equations; generalizations, such as the introduction of subtyping, are not easy. Furthermore, Algorithm $\mathcal{J}$ works with *substitutions*, instead of *constraints*. Substitutions are an approximation to solved forms for unification constraints. Working with substitutions means using *most general unifiers*, *composition*, and *restriction*. Working with constraints means using *equations*, *conjunction*, and *existential quantification*.

$$
\begin{array}{rcl}
\mathcal{J}(\Gamma \vdash x) & = & \mathsf{let}\ \forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
& & \mathsf{do}\ \alpha'_1, \ldots, \alpha'_n = \mathsf{fresh}, \ldots, \mathsf{fresh} \\
& & \mathsf{return}\ [\alpha_i \mapsto \alpha'_i]_{i=1}^n(\tau)\ \text{-- } \textit{take a fresh instance} \\
\mathcal{J}(\Gamma \vdash \lambda x.\, a_1) & = & \mathsf{do}\ \alpha = \mathsf{fresh} \\
& & \mathsf{do}\ \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash a_1) \\
& & \mathsf{return}\ \alpha \to \tau_1\ \text{-- } \textit{form an arrow type} \\
\mathcal{J}(\Gamma \vdash a_1\ a_2) & = & \mathsf{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
& & \mathsf{do}\ \tau_2 = \mathcal{J}(\Gamma \vdash a_2) \\
& & \mathsf{do}\ \alpha = \mathsf{fresh} \\
& & \mathsf{do}\ \varphi \leftarrow \mathsf{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \to \alpha)) \circ \varphi \\
& & \mathsf{return}\ \alpha\ \text{-- } \textit{solve}\ \tau_1 = \tau_2 \to \alpha \\
\mathcal{J}(\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2) & = & \mathsf{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
& & \mathsf{let}\ \sigma = \forall\!\smallsetminus\!\mathsf{ftv}(\varphi(\Gamma)).\, \varphi(\tau_1)\ \text{-- } \textit{generalize} \\
& & \mathsf{return}\ \mathcal{J}(\Gamma; x : \sigma \vdash a_2)
\end{array}
$$

($\forall\!\smallsetminus\!\bar{\alpha}.\tau$ quantifies over all type variables *other than* $\bar{\alpha}$.)

Figure 5.3: Type inference algorithm for ML

5.3.2 Constraint-based type inference for ML

Type inference for Damas and Milner's type system involves slightly more than first-order unification: there is also *generalization* and *instantiation* of type schemes. So, the constraint language must be enriched. We proceed in two steps: still within simply-typed λ-calculus, we present a variation of the constraint language; building on this variation, we introduce polymorphism.

How about letting the constraint solver, instead of the constraint generator, deal with *environment access* and *construction?* That is, the syntax of constraints is as follows:

$$C ::= \ldots \mid x = \tau \mid \mathsf{def}\ x : \tau\ \mathsf{in}\ C$$

The idea is to interpret constraints in such a way as to validate the equivalence law:

$$\mathsf{def}\ x : \tau\ \mathsf{in}\ C \quad \equiv \quad [x \mapsto \tau]C$$

The def form is an *explicit substitution* form. More precisely, here is the new interpretation of constraints. As before, a valuation ϕ maps type variables α to ground types. In addition, a valuation ψ maps term variables x to ground types. The satisfaction judgment now takes the form $\phi, \psi \vdash C$. The new rules of interest are:

$$\frac{\psi x = \phi\tau}{\phi, \psi \vdash x = \tau} \qquad\qquad \frac{\phi, \psi[x \mapsto \phi\tau] \vdash C}{\phi, \psi \vdash \mathsf{def}\ x : \tau\ \mathsf{in}\ C}$$

(All other rules are modified to just transport ψ.) Constraint generation becomes a mapping of an expression a and a type τ to a constraint $\langle\!\langle a : \tau \rangle\!\rangle$. There is no longer a need for the

$$
\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x = \tau \\
\langle\!\langle \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\mathsf{def}\ x : \alpha_1\ \mathsf{in}\ \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
& & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
& & \text{if } \alpha \# a_1, a_2, \tau
\end{array}
$$

Figure 5.4: Constraints with program variables

parameter Γ. Constraint generation is defined in Figure 5.4

Theorem 16 (Soundness and completeness) *Assume* $\mathsf{fv}(a) = \mathsf{dom}(\Gamma)$. *Then,* $\phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

Corollary 32 *Assume* $\mathsf{fv}(a) = \varnothing$. *Then, a is well-typed if and only if* $\exists \alpha.\langle\!\langle a : \alpha \rangle\!\rangle \equiv \mathsf{true}$.

This variation shows that there is *freedom* in the design of the constraint language, and that altering this design can *shift work* from the constraint generator to the constraint solver, or vice-versa.

Enriching constraints To permit polymorphism, we must extend the syntax of constraints so that a variable x denotes not just a ground type, but a *set of ground types*.

However, these sets cannot be represented as type schemes $\forall \bar{\alpha}.\tau$, because constructing these simplified forms requires constraint solving. To avoid mingling constraint generation and constraint solving, we use type schemes that incorporate constraints, called *constrained type schemes*. The syntax of *constraints* and of *constrained type schemes* is:

$$
\begin{array}{rcl}
C & ::= & \tau = \tau \mid C \wedge C \mid \exists \alpha.C \mid x \le \tau \mid \sigma \le \tau \mid \mathsf{def}\ x : \sigma\ \mathsf{in}\ C \\
\sigma & ::= & \forall \bar{\alpha}[C].\,\tau
\end{array}
$$

Both $x \le \tau$ and $\sigma \le \tau$ are *instantiation constraints*. The latter form is introduced so as to make the syntax stable under substitutions of constrained type schemes for variables. As before, $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ is an *explicit substitution* form.

The idea is to interpret constraints in such a way as to validate the equivalence laws:

$$
\mathsf{def}\ x : \sigma\ \mathsf{in}\ C \;\equiv\; [x \mapsto \sigma]C \qquad (\forall \bar{\alpha}[C].\,\tau) \le \tau' \;\equiv\; \exists \bar{\alpha}.(C \wedge \tau = \tau') \quad \text{if } \bar{\alpha} \# \tau'
$$

Using these laws, a closed constraint can be rewritten to a unification constraint (with a possibly exponential increase in size). The new constructs do not add much expressive power. They add just enough to allow a stand-alone formulation of constraint generation.

The interpretation of constraints must be redefined since the environment ψ now maps program variables to sets of ground types. The environment ϕ still maps type variables to ground types. Hence, a type variable α still denotes a ground type. A variable x now denotes

a *set* of ground types. Instantiation constraints are interpreted as *set membership.* The rules for the new form of constraints are:

$$\frac{\phi\tau \in \psi x}{\phi, \psi \vdash x \leq \tau} \qquad \frac{\phi\tau \in \binom{\phi}{\psi}\sigma}{\phi, \psi \vdash \sigma \leq \tau} \qquad \frac{\phi, \psi[x \mapsto \binom{\phi}{\psi}\sigma] \vdash C}{\phi, \psi \vdash \mathsf{def}\ x : \sigma\ \mathsf{in}\ C}$$

The interpretation of $\forall\bar{\alpha}[C].\,\tau$ under ϕ and ψ, written $\binom{\phi}{\psi}(\forall\bar{\alpha}[C].\,\tau)$ is the set of all $\phi'\tau$, where ϕ and ϕ' coincide outside $\bar{\alpha}$ and where ϕ' and ψ satisfy C:

$$\binom{\phi}{\psi}(\forall\bar{\alpha}[C].\,\tau) \ \stackrel{\triangle}{=}\ \{\phi'\tau \mid (\phi' \setminus \bar{\alpha} = \phi \setminus \bar{\alpha}) \wedge (\phi', \psi \vdash C)\}$$

If C is empty, then $\binom{\phi}{\psi}(\forall\bar{\alpha}[C].\,\tau)$ is $\{(\phi[\bar{\alpha} \mapsto \bar{\mathbf{t}}])\tau\}$. If $\bar{\alpha}$ and C are empty, then $\binom{\phi}{\psi}\tau$ is $\phi\tau$.

For instance, the interpretation of $\forall\alpha[\exists\beta.\alpha = \beta \to \gamma].\,\alpha \to \alpha$ under ϕ and ψ is the set of all ground types of the form $(t \to \phi\gamma) \to (t \to \phi\gamma)$, where t ranges over ground types. This is also the interpretation of an unconstrained typed scheme, namely $\forall\beta.\,(\beta \to \gamma) \to (\beta \to \gamma)$. In fact, this is a general situation:

Lemma 33 *Every constrained type scheme is equivalent to a standard type scheme.*

This result holds because constraints can be reduced to unification constraints, which have either no solution or a principal solution. This is an important property as it implies that type inference problems have principal solutions and typable programs have principal types. The property would not hold with more general constraints, such as subtyping constraints. However, we may then generalize type schemes to constrained type schemes as a way to factor several possible types and recover principality of type inference. Then, type inference may have principal *constrained type schemes.*

Notice that if x does not appear free in C, $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ is equivalent to C—whether or not the constraints appearing in σ are solvable. To enforce the constraints in σ to be solvable, we use a variant of the def construct:

$$\mathsf{let}\ x : \sigma\ \mathsf{in}\ C \ \stackrel{\triangle}{=}\ \mathsf{def}\ x : \sigma\ \mathsf{in}\ ((\exists\alpha.x \leq \alpha) \wedge C)$$

Expanding the constraint type scheme σ of the form $\forall\bar{\alpha}[C].\,\tau$ and simplifying, an equivalent definition is:

$$\mathsf{let}\ x : \forall\bar{\alpha}[C].\,\tau\ \mathsf{in}\ C' \ \stackrel{\triangle}{=}\ \exists\bar{\alpha}.C \wedge \mathsf{def}\ x : \forall\bar{\alpha}[C].\,\tau\ \mathsf{in}\ C'$$

This is equivalent to providing a direct interpretation of let-bindings as:

$$\frac{\binom{\phi}{\psi}\sigma \neq \varnothing \qquad \phi, \psi[x \mapsto \binom{\phi}{\psi}\sigma] \vdash C}{\phi, \psi \vdash \mathsf{let}\ x : \sigma\ \mathsf{in}\ C}$$

Constraint generation for ML is defined in Figure 5.5. The abbreviation $(\!| a |\!)$ is a *principal constrained type scheme* for a: its intended interpretation is the set of all ground types that a admits.

$$
\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x \preceq \tau \\
\langle\!\langle \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\mathsf{def}\ x : \alpha_1\ \mathsf{in}\ \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# a_1, a_2, \tau \\
\langle\!\langle \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau \rangle\!\rangle & = & \mathsf{let}\ x : (\!| a_1 |\!)\ \mathsf{in}\ \langle\!\langle a_2 : \tau \rangle\!\rangle \\
(\!| a |\!) & = & \forall \alpha\big[\langle\!\langle a : \alpha \rangle\!\rangle\big].\, \alpha
\end{array}
$$

Figure 5.5: Constraint generation for ML

Lemma 34 (Constraint equivalences) *The following equivalences hold:*

(1)	$\exists \alpha.(\langle\!\langle a : \alpha \rangle\!\rangle \wedge \alpha = \tau)$	$\equiv$	$\langle\!\langle a : \tau \rangle\!\rangle$	*if* $\alpha \# \tau$
(2)	$(\!| a |\!) \preceq \tau$	$\equiv$	$\langle\!\langle a : \tau \rangle\!\rangle$	
(3)	$[x \mapsto (\!| a_1 |\!)]\langle\!\langle a_2 : \tau \rangle\!\rangle$	$\equiv$	$\langle\!\langle [x \mapsto a_1]a_2 : \tau \rangle\!\rangle$	

Proof: (1) is by induction on the definition of $\langle\!\langle a : \tau \rangle\!\rangle$; (2) is by definition of $(\!| a |\!)$, expansion of the instantiation constraint and (1); (3) is by induction on $\langle\!\langle a : \tau \rangle\!\rangle$ and (2).

Another key property is that the constraint associated with a let construct is *equivalent* to the constraint associated with its let-normal form.

Lemma 35 (let expansion) $\langle\!\langle \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau \rangle\!\rangle \ \equiv\ \langle\!\langle a_1; [x \mapsto a_1]a_2 : \tau \rangle\!\rangle$.

Expansion of let-binding terminates, since it can be seen as reducing the family of redexes marked as let-bindings. The resulting expression has no let-binding and its constraint has no def-constraint. Hence, its interpretation is the same as constraints for the simply-typed λ-calculus. This gives another specification of ML: *a closed program is well-typed in ML if and only if its let-expansion is typable with simple types.*

Constraint generation for ML can still be implemented in linear time and space.

Lemma 36 *The size of* $\langle\!\langle a : \tau \rangle\!\rangle$ *is linear in the sum of the sizes of* a *and* τ.

The statement of soundness and completeness keeps its previous form, but Γ now contains Damas-Milner type schemes. Since Γ binds variables to type schemes, we define $\phi(\Gamma)$ as the point-wise mapping of $\binom{\phi}{\varnothing}$ to Γ.

Theorem 17 (Soundness and completeness) *Assume* $\mathsf{fv}(a) = \mathsf{dom}(\Gamma)$. *Then,* $\phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

Key points Notice that constraint generation has *linear complexity*; constraint generation and constraint solving are *separate*. This makes constraints suitable for use in an efficient and modular implementation. In particular, the constraint language remains *small* as the programming language grows.

5.3.3 Constraint solving by example

For our running example, assume that the *initial environment* Γ_0 stands for $assoc : \forall\alpha\beta.\alpha \to \mathsf{list}\ (\alpha \times \beta) \to \beta$. That is, the constraints considered next are implicitly wrapped within the context $\mathsf{def}\ \Gamma_0\ \mathsf{in}\ []$. Let a stand for the term:

$$\lambda x.\lambda l_1.\lambda l_2.\mathsf{let}\ assocx = assoc\ \ x\ \mathsf{in}\ (assocx\ \ l_1, assocx\ \ l_2)$$

One may anticipate that $assocx$ receives a polymorphic type scheme, which is instantiated twice at different types. Let Γ stand for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$. Then, the constraint $\langle\!\langle a : \alpha \rangle\!\rangle$ is, after a few minor simplifications:

$$\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathsf{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ assocx : \forall\gamma_1\left[\exists\gamma_2.\left(\begin{array}{l} assoc \le \gamma_2 \to \gamma_1 \\ x \le \gamma_2 \end{array}\right)\right].\gamma_1\ \mathsf{in} \\ \quad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\ \ \exists\gamma_2.(assocx \le \gamma_2 \to \beta_i \wedge l_i \le \gamma_2) \end{array}\right) \end{array}\right)$$

Constraint solving can be viewed as a *rewriting process* that exploits *equivalence laws*. Because equivalence is, by construction, a *congruence*, rewriting is permitted within an arbitrary context. For instance, environment access is allowed by the law

$$\mathsf{let}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[x \le \tau] \quad \equiv \quad \mathsf{let}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[\sigma \le \tau]$$

where $\mathcal{R}$ is a context that does not bind x. Thus, within the context $\mathsf{def}\ \Gamma_0; \Gamma\ \mathsf{in}\ []$, we have the following equivalence:

$$assoc \le \gamma_2 \to \gamma_1\ \wedge\ x \le \gamma_2 \quad \equiv \quad \exists\alpha\beta.(\alpha \to \mathsf{list}\ (\alpha \times \beta) \to \beta = \gamma_2 \to \gamma_1)\ \wedge\ \alpha_0 = \gamma_2$$

By first-order unification, we have the following sequence of simplifications:

$$\begin{array}{rl} & \exists\gamma_2.(\exists\alpha\beta.\quad (\alpha \to \mathsf{list}\ (\alpha\ \times\ \beta) \to \beta = \gamma_2 \to \gamma_1) \wedge \alpha_0 = \gamma_2) \\ \equiv & \exists\gamma_2.(\exists\alpha\beta.(\alpha = \gamma_2 \wedge \mathsf{list}\ (\alpha\ \times\ \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2) \\ \equiv & \exists\gamma_2.(\exists\beta.(\mathsf{list}\ (\gamma_2 \times \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2) \\ \equiv & \exists\beta.(\mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_1) \end{array}$$

Hence,

$$\begin{array}{rcl} \forall\gamma_1[\exists\gamma_2.(assoc \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2)].\gamma_1 & \equiv & \forall\gamma_1[\exists\beta.(\mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_1)].\gamma_1 \\ & \equiv & \forall\gamma_1\beta[\mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_1].\gamma_1 \\ & \equiv & \forall\beta.\mathsf{list}\ (\alpha_0 \times \beta) \to \beta \end{array}$$

We have used the rule:

$$\forall\alpha[\exists\beta.C].\tau \equiv \forall\alpha\beta[C].\tau \qquad \text{if } \beta \# \tau$$

The initial constraint has now been simplified down to:

$$\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l}\alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathsf{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{assocx} : \forall\beta.\,\mathsf{list}\ (\alpha_0 \times \beta) \to \beta\ \mathsf{in} \\ \quad \exists\beta_1\beta_2.\left(\begin{array}{l}\beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\ \ \exists\gamma_2.(\mathit{assocx} \preceq \gamma_2 \to \beta_i \wedge l_i \preceq \gamma_2)\end{array}\right)\end{array}\right)$$

The simplification work spent on *assocx*'s type scheme was well worth the trouble, because we are now going to *duplicate* the simplified type scheme.

The subconstraint $\exists\gamma_2.(\mathit{assocx} \preceq \gamma_2 \to \beta_i \wedge l_i \preceq \gamma_2)$ where $i \in \{1,2\}$, is rewritten:

$$\begin{array}{ll} & \exists\gamma_2.(\exists\beta.(\mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_2 \to \beta_i) \wedge \alpha_i = \gamma_2) \\ \equiv & \exists\beta.(\mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \alpha_i \to \beta_i) \\ \equiv & \exists\beta.(\mathsf{list}\ (\alpha_0 \times \beta) = \alpha_i \wedge \beta = \beta_i) \\ \equiv & \mathsf{list}\ (\alpha_0 \times \beta_i) = \alpha_i \end{array}$$

The initial constraint has now been simplified down to:

$$\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l}\alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathit{assocx} : \forall\beta.\,\mathsf{list}\ (\alpha_0 \times \beta) \to \beta\ \mathsf{in}\ \exists\beta_1\beta_2.\left(\begin{array}{l}\beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\}, \mathsf{list}\ (\alpha_0 \times \beta_i) = \alpha_i\end{array}\right)\end{array}\right)$$

Now, the context $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathit{assocx} : \ldots\ \mathsf{in}\ []$ can be dropped, because the constraint that it applies to contains no occurrences of x, l_1, l_2, or *assocx*. The constraint becomes:

$$\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l}\alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \exists\beta_1\beta_2.\left(\begin{array}{l}\beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\ \ \mathsf{list}\ (\alpha_0 \times \beta_i) = \alpha_i\end{array}\right)\end{array}\right)$$

that is, by extrusion:

$$\exists\alpha_0\alpha_1\alpha_2\beta\beta_1\beta_2.\left(\begin{array}{l}\alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\ \ \mathsf{list}\ (\alpha_0 \times \beta_i) = \alpha_i\end{array}\right)$$

Finally, by eliminating a few auxiliary variables:

$$\exists\alpha_0\beta_1\beta_2.(\alpha = \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta_1) \to \mathsf{list}\ (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2)$$

We have shown the following equivalence between constraints:

$$\mathsf{def}\ \Gamma_0\ \mathsf{in}\ \langle\!\langle a : \alpha \rangle\!\rangle \quad\equiv\quad \exists\alpha_0\beta_1\beta_2.(\alpha = \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta_1) \to \mathsf{list}\ (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2)$$

That is, the *principal type scheme* of a relative to Γ_0 is

$$(\!| a |\!) \quad=\quad \forall\alpha\big[\langle\!\langle a : \alpha \rangle\!\rangle\big].\alpha \quad\equiv\quad \forall\alpha_0\beta_1\beta_2.\ \ \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta_1) \to \mathsf{list}\ (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2$$

Again, constraint solving can be explained in terms of a *small-step rewrite system.* Again, one checks that every step is meaning-preserving, that the system is normalizing, and that every normal form is either literally "false" or satisfiable.

Rewriting strategies Different constraint solving *strategies* lead to different behaviors in terms of complexity, error explanation, etc. See Pottier and Rémy (2005) for details on constraint solving. See Jones (1999b) for a different presentation of type inference, in the context of Haskell.

In all reasonable strategies, the left-hand side of a let constraint is simplified *before* the let form is expanded away. This corresponds, in Algorithm $\mathcal{J}$, to computing a principal type scheme before examining the right-hand side of a let construct.

Complexity Type inference for ML is DEXPTIME-complete (Kfoury et al., 1990; Mairson, 1990), so any constraint solver has exponential complexity. This is assuming that types are printed as trees. If one allows to return types are dags graphs instead of types, the complexity is EXPTIME-complete.

This is, of course, worse case complexity, which does not contradict the observation that ML type inference *works well in practice.*

If fact, this good behavior can be explain by the results of McAllester (2003): under the hypotheses that *types have bounded size* and let forms have bounded left-nesting depth, constraints can be solved in linear time, or in quasi-linear time if recursive types are allowed.

When the size of types in unbounded, one may reach worst case complexity but right-nesting let-bindings as in Mairson original example:

$$
\begin{array}{l}
\texttt{let}\ \mathit{mairson} = \\
\quad \texttt{let}\ f = \texttt{fun}\ x \to (x, x)\ \texttt{in} \\
\quad \textit{(* ... n times ... *)} \\
\quad \texttt{let}\ f = \texttt{fun}\ x \to f\,(f\,x)\ \texttt{in} \\
\quad f\ (\texttt{fun}\ z \to z)
\end{array}
$$

This term can be placed in the context $\texttt{let}\ x = ...\ \texttt{in}\ ()$ to ignore the time spent outputing the result type.

However, this right-nesting of let-bindings is not a problem if types remain bounded, because each let-bound expression can be simplified to a type of bounded size before being duplicated.

On the opposite, in a left-nesting of let-binding local variables may have to be extruded step by step from the inner bindings to its enclosing binding, sometimes all the way up to the root, leading to a quadratic complexity when the nesting is proportional to the size of the program.

Principal constraint type schemes In constraint generation, we introduced principal constraint type scheme $(\!| a |\!)$ as an abbreviation for $\forall\alpha\big[\langle\!\langle a : \alpha \rangle\!\rangle\big].\,\alpha$. However, using the equiv-

$$
\begin{array}{rcl}
(\!| x |\!) & = & \forall\alpha[x \le \alpha].\,\alpha \\
(\!| \lambda x.\, a |\!) & = & \forall\alpha_1\alpha_2\big[\mathsf{def}\ x : \alpha_2\ \mathsf{in}\ (\!| a |\!) \le \alpha_1\big].\,\alpha_2 \to \alpha_1 \\
& & \text{if } \alpha_1, \alpha_2 \# a \\
(\!| a_1\ a_2 |\!) & = & \forall\alpha_1\alpha_2\big[(\!| a_1 |\!) \le \alpha_2 \to \alpha_1 \wedge (\!| a_2 |\!) \le \alpha_2\big].\,\alpha_1 \\
& & \text{if } \alpha_1, \alpha_2 \# a_1, a_2 \\
(\!| \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 |\!) & = & \forall\alpha\big[\mathsf{let}\ x : (\!| a_1 |\!)\ \mathsf{in}\ (\!| a_2 |\!) \le \alpha\big].\,\alpha
\end{array}
$$

Figure 5.6: Constraint generation with principal constraint type schemes

alence between $\langle\!\langle a : \tau \rangle\!\rangle$ and $(\!| a |\!) \le \tau$, we may conversely use principal constraint type schemes in place of program constraints. This leads to an alternative presentation of constraint generation described in Figure 5.6. (Compare it with the previous definition in Figure 5.5).

5.3.4 Type reconstruction

Type inference should not just return a principal type for an expression; it should also perform type reconstruction, *i.e.* elaborate the implicitly-typed input term into an explicitly-typed one.

The elaborated term is not unique, since redundant type abstractions and type applications may always be used. Moreover, some non principal type schemes may also be used for local let-bindings—even if the final type is principal.

For example the implicitly-typed term $\mathsf{let}\ x = \lambda y.\, y\ \mathsf{in}\ x\ 1$ may be explicitly typed as either one of

$$\mathsf{let}\ x : \mathsf{int} \to \mathsf{int} = \lambda y{:}\mathsf{int}.\, y\ \mathsf{in}\ x\ 1 \qquad\qquad \mathsf{let}\ x : \forall\alpha.\,\alpha \to \alpha = \Lambda x.\lambda x{:}\mathsf{int}.\, x\ \mathsf{in}\ x\ \mathsf{int}\ 1$$

Which one is better? Monomorphic terms can be compiled more efficiently, so removing useless polymorphism may be useful.

However, one usually infers more general explicitly-typed terms. Given explicitly-typed terms M and M' with the same type erasure, we say that M is *more general* than M' if all let-bindings are assigned more general type schemes in M than in M', *i.e.*:

> for all decompositions of M into $C[\mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2]$, then there is a corresponding decomposition of M' (*i.e.* one where C and C' have the same erasure) as $C'[\mathsf{let}\ x : \sigma' = M_1'\ \mathsf{in}\ M_2']$ where σ is more general than σ'.

A *type reconstruction is principal* if it is more general than any other type reconstruction of the same term. *Core ML* admits principal type reconstructions. A principal typing derivation can be seeked for in canonical form, as defined in 4.6.2.

A term in canonical form is uniquely determined up to reordering of type abstractions and type applications by the type schemes of bound program variables and of how they are

instanced. We may keep track of such information during constraint resolution by keeping the binding constraints $\mathsf{def}\ x : C\ \mathsf{in}\ C$ and its derived form $\mathsf{let}\ x : C\ \mathsf{in}\ C$, and the instantiation constraints $x \le \tau$ of the original constraint—instead of removing them once solved. We call them *persistent constraints*. We thus forbid the removal, as well as the extrusion of persistent constraints by restricting the equivalence of constraints accordingly.

Rewriting rules used for constraint resolution can easily be adapted to retain the persistent constraints—and thus preserve the restricted notion of equivalence. Then, the binding structure of the constraint remains unchanged during simplification and is isomorphic to the binding structure of the expression it came from. (Persistent nodes could actually be labeled by their corresponding nodes in the original expression.)

In practice, we mark nodes of the persistent constraints as *resolved* when they could have been dropped in the normal resolution process—so that they need not be considered anymore during the resolution. For example, we use the rule

$$\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[x \le \tau] \quad \equiv \quad \mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[\color{blue}{x \le \tau} \color{black}{\wedge \sigma \le \tau}]$$

for environment access, where the original constraint $\color{blue}{x \le \tau}$ is kept and marked as resolved but is not removed. Similarly, a constraint $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ can be marked as resolved, which we write $\color{blue}{\mathsf{def}\ x : \sigma\ \mathsf{in}}\ C$, whenever x may only appears free in removable constraints of C. A resolved form of a constraint is an equivalent persistent constraint, such that dropping all persistent nodes is an equivalent constraint in solved forms.

For example, reusing the running example and notations of the previous section, let us find a term M whose erasure a is defined as:

$$\lambda x.\, \lambda l_1.\, \lambda l_2.\, \mathsf{let}\ \mathit{assocx} = \mathit{assoc}\ \ x\ \mathsf{in}\ (\mathit{assocx}\ \ l_1, \mathit{assocx}\ \ l_2)$$

The principal type scheme $(\!| a |\!)$ is, by definition:

$$\forall \alpha \left[\exists \alpha_0 \alpha_1 \alpha_2 \beta. \left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathsf{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{assocx} : \forall \gamma_1 \left[\exists \gamma_2. \left(\begin{array}{l} \mathit{assoc} \le \gamma_2 \to \gamma_1 \\ x \le \gamma_2 \end{array} \right) \right]. \gamma_1\ \mathsf{in} \\ \quad \exists \beta_1 \beta_2. \left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1, 2\},\ \ \exists \gamma_2.(\mathit{assocx} \le \gamma_2 \to \beta_i \wedge l_i \le \gamma_2) \end{array} \right) \end{array} \right) \right]. \alpha$$

Since $x : \alpha_9$ is in Γ, the inner constraint can be resolved as follows:

$$\begin{array}{ll} & \exists \gamma_2.(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2) \\ \equiv & \exists \gamma_2.(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge \color{blue}{x \le \gamma_2} \color{black}{\wedge \alpha_0 \le \gamma_2)} \quad \equiv \quad \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge \color{blue}{x \le \alpha_0} \end{array}$$

The other instantiation may be solved similarly, leading to the equivalent constraints:

$$\begin{array}{ll} & \color{blue}{\mathit{assoc} \le \alpha_0 \to \gamma_1} \color{black}{\wedge \forall \alpha\beta.\, \alpha \to \mathsf{list}\ (\alpha \times \beta) \to \beta \le \alpha_0 \to \gamma_1 \wedge} \color{blue}{x \le \alpha_0} \\ \equiv & \color{blue}{\mathit{assoc} \le \alpha_0 \to \gamma_1} \color{black}{\wedge \exists \alpha\beta.(\alpha = \alpha_0 \wedge \mathsf{list}\ (\alpha \times \beta) \to \beta = \gamma_1) \wedge} \color{blue}{x \le \alpha_0} \\ \equiv & \exists \beta.(\color{blue}{\mathit{assoc} \le \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta) \to \beta} \color{black}{\wedge \mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_1 \wedge} \color{blue}{x \le \alpha_0}\color{black}{)} \end{array}$$

Hence, the type scheme of assoc is equivalent to

$$\forall\beta[\mathit{assoc} \leq \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta) \to \beta \wedge x \leq \alpha_0].\ \mathsf{list}\ (\alpha_0 \times \beta) \to \beta$$

and $(\!|\, a_1 \,|\!)$ is equivalent to:

$$\forall\alpha\left[\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathsf{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{assocx} : \forall\beta\left[\ \mathit{assoc} \leq \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta) \to \beta \wedge x \leq \alpha_0\ \right]. \\ \hfill \mathsf{list}\ (\alpha_0 \times \beta) \to \beta\ \mathsf{in} \\ \quad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\quad \exists\gamma_2.(\mathit{assocx} \leq \gamma_2 \to \beta_i \wedge l_i \leq \gamma_2) \end{array}\right) \end{array}\right)\right].\ \alpha$$

Simplifying the remaining instantiation constraints in a similar way, we end up with the following resolved type scheme for $(\!|\, a \,|\!)$:

$$\forall\alpha_0\beta_1\beta_2\left[\begin{array}{l} \mathsf{def}\ \Gamma\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{assocx} : \forall\gamma\left[\begin{array}{l} \mathit{assoc} \leq \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \gamma) \to \gamma \\ x \leq \alpha_0 \end{array}\right].\ \mathsf{list}\ (\alpha_0 \times \gamma) \to \gamma\ \mathsf{in} \\ \quad \forall i \in \{1,2\},\quad \mathit{assocx} \leq \mathsf{list}\ (\alpha_0 \times \beta_i) \to \beta_i \wedge l_i \leq \mathsf{list}\ (\alpha_0 \times \beta_i) \end{array}\right].$$
$$\alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta_1) \to \mathsf{list}\ (\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2$$

This is a resolved form, from which we may build the elaboration of a_1:

$$\begin{array}{l} \Lambda\alpha_0\beta_1\beta_2.\lambda x{:}\alpha_0.\ \lambda l_1{:}\mathsf{list}\ (\alpha_0 \times \beta_1).\ \lambda l_2{:}\mathsf{list}\ (\alpha_0 \times \beta_2). \\ \qquad\qquad \mathsf{let}\ \mathit{assocx} = \Lambda\gamma.\mathit{assoc}\ \alpha_0\ \gamma\ \ x\ \mathsf{in}\ (\mathit{assocx}\ \beta_1\ \ l_1, \mathit{assocx}\ \beta_2\ \ l_2) \end{array}$$

Type abstractions are determined by their corresponding type scheme in the resolved constraint; for instance, the type abstraction for the let-bound variable *assocx* is γ while the toplevel type abstraction is $\alpha_0\alpha_1\beta_2$. Type annotations on abstractions are determined by Γ, which here contains $x : \alpha_0; l_1 : \mathsf{list}\ (\alpha_0 \times \alpha_1); l_2 : \mathsf{list}\ (\alpha_0 \times \alpha_2)$. Type applications are inferred locally by looking at their corresponding type instantiations in the resolved constraints. For instance, we read from the constraint that *assocx* is let-bound with the type scheme $\forall\gamma.\ \mathsf{list}\ (\alpha_0 \times \gamma) \to \gamma$ (we dropped the constraint which is solved and equivalent to true) and that its i-th occurrence is used at type $\mathsf{list}\ (\alpha_0 \times \beta_i) \to \beta_i$. Matching the former against the latter gives the substitution $\gamma \mapsto \beta_i$. Thefefore, the type application for the i's occurrence is be β_i.

Modular type reconstruction One criticism of our appoach is that the mecanism for type reconstruction is based on *program* typing constraints and not on *type* constraint alone. Hence, we do not have a clear separation of separation of concerns. Modularity can be achieved by defining for each construct of the language taken independently the constraint generation together with the elaboration of this construct once the constraint will have been solved. See Pottier (2014) for details.

Principal type reconstruction Notice that while the constraint framework enforces the inference of principal types, since it transforms the original constraint into an *equivalent constraint*, it does not enforce type reconstruction to be principal. Indeed, in a constraint $\exists\alpha. C$, the existentially bound type variable α may be instantiated to *any* type that satisfies the constraint C and not necessarily the most general one.

Interestingly, however, the default *strategy* for constraint resolution always *returns principal type reconstructions.* That is, variables are never arbitrarily instantiated, although this would be allowed by the specification.

Exercise 36 (Minimal derivations) *On the opposite, one may seek for less general typing derivations where all let-expressions are as instantiated as possible. Do such derivations exists? In fact no: there are examples where there are two minimal incomparable type reconstructions and others with smaller and smaller type reconstructions but no smallest one. Find examples of both kinds.* *(Solution p. 125)* □

Exercise 37 (Closed types) *Explain why ML modules in combination with the value-restriction break the principal type property: that is, there are programs that are typable but that do not have a principal type. Hint: ML signatures of ML modules must be closed.* *(Solution p. 125)* □

5.4 Type annotations

Damas and Milner's type system has *principal types:* at least in the core language, no type information is required. This is very lightweight, but a bit extreme: sometimes, it is useful to write types down, and use them as *machine-checked documentation.* Let us, then, allow programmers to *annotate* a term with a type:

$$a ::= \ldots \mid (a : \tau)$$

Typing and constraint generation are obvious:

$$\frac{\textsc{Annot}\quad \Gamma \vdash a : \tau}{\Gamma \vdash (a : \tau) : \tau} \qquad\qquad \langle\!\langle (a : \tau) : \tau' \rangle\!\rangle = \langle\!\langle a : \tau \rangle\!\rangle \wedge \tau = \tau'$$

Type annotations are *erased* prior to runtime, so the operational semantics is not affected. In particular, it is still type-erasing.

Notice that annotations here do not help type more terms, as erasure of type annotations preserves well-typedness: Indeed, the constraint $\langle\!\langle (a : \tau) : \tau' \rangle\!\rangle$ *implies* the constraint $\langle\!\langle a : \tau' \rangle\!\rangle$. That is, in terms of type inference, *type annotations are restrictive* : they lead to a principal type that is less general, and possibly even to ill-typedness. For instance, $\lambda x. x$ has principal type scheme $\forall\alpha. \alpha \to \alpha$, whereas $(\lambda x. x : \mathsf{int} \to \mathsf{int})$ has principal type scheme $\mathsf{int} \to \mathsf{int}$, and $(\lambda x. x : \mathsf{int} \to \mathsf{bool})$ is ill-typed.

5.4.1 Explicit binding of type variables

We must be careful with type variables within type annotations, as in, say:

$$(\lambda x.\, x : \alpha \to \alpha) \qquad (\lambda x.\, x + 1 : \alpha \to \alpha) \qquad \mathsf{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true})$$

Does it make sense, and is so, what does it mean? A short answer is that *it does not mean anything, because* α *is unbound.* "There is no such thing as a free variable" *(Alan Perlis).* A longer answer is that *it is necessary to specify* how *and* where *variables are bound.*

How is α bound? If α is *existentially* bound, or *flexible,* then both $(\lambda x.\, x : \alpha \to \alpha)$ and $(\lambda x.\, x + 1 : \alpha \to \alpha)$ should be well-typed. If it is *universally* bound, or *rigid,* only the former should be well-typed.

Where is α bound? If α is bound *within* the left-hand side of this "let" construct, then $\mathsf{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true})$ should be well-typed. On the other hand, if α is bound *outside* this "let" form, then this code should be ill-typed, since no *single* ground value of α is suitable.

Programmers should *explicitly bind* type variables. We extend the syntax of expressions as follows:

$$a ::= \ldots \mid \exists \bar\alpha.a \mid \forall \bar\alpha.a$$

It now makes sense for a type annotation $(a : \tau)$ to contain free type variables—as long as these type variables have been introduced in some enclosing term.

Since terms can now contain free type variables, some side conditions have to be updated (e.g., $\bar\alpha \# \Gamma, a$ in GEN). The new (and updated) typing rules are as follows:

$$\frac{\Gamma \vdash [\vec\alpha \mapsto \vec\tau]a : \tau}{\Gamma \vdash \exists \bar\alpha.a : \tau}\ \textsc{Exists} \qquad \frac{\Gamma \vdash a : \tau \qquad \bar\alpha \# \Gamma}{\Gamma \vdash \forall \bar\alpha.a : \forall \bar\alpha.\tau}\ \textsc{Forall} \qquad \left(\frac{\Gamma \vdash a : \tau \qquad \bar\alpha \# \Gamma, a}{\Gamma \vdash a : \forall \bar\alpha.\tau}\ \textsc{Gen}\right)$$

As type annotations, the introduction of type variables are erased prior to runtime.

Exercise 38 *Define the erasure of implicitly-typed terms and show that the erasure of a well-typed term is well-typed. Use this to justify the soundness of the extension of* ML *with type annotations with explicit introduction of type variables.* □

Constraint generation for the existential form is straightforward:

$$\langle\!\langle (\exists \bar\alpha.a) : \tau \rangle\!\rangle \;=\; \exists \bar\alpha.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar\alpha \# \tau$$

The type annotations inside a contain free occurrences of $\bar\alpha$. Thus, the constraint $\langle\!\langle a : \tau \rangle\!\rangle$ contains such occurrences as well, which are bound by the existential quantifier.

For example, the expression $\lambda x_1.\, \lambda x_2.\, \exists \alpha.((x_1 : \alpha), (x_2 : \alpha))$ has principal type scheme $\forall \alpha.\, \alpha \to \alpha \to \alpha \times \alpha$. Indeed, the generated constraint is of the form $\exists \alpha.(\langle\!\langle x_1 : \alpha \rangle\!\rangle \wedge \langle\!\langle x_2 : \alpha \rangle\!\rangle \wedge \ldots)$, which requires x_1 and x_2 to *share* a common (unspecified) type.

Perhaps surprisingly, constraint generation for the universal case is more difficult. A term a has type scheme, say, $\forall \alpha.\, \alpha \to \alpha$ if and only if a has type $\alpha \to \alpha$ *for every instance of*

α, or, equivalently, for an abstract α. To express this in terms of constraints, we introduce *universal quantification* in the constraint language:

$$C ::= \ldots \mid \forall\alpha.C$$

Its interpretation is as expected:

$$\frac{\forall \mathbf{t}, \quad \phi[\alpha \mapsto \mathbf{t}], \psi \vdash C}{\phi, \psi \vdash \forall\alpha.C}$$

(To solve these constraints, we will use an extension of the unification algorithm called unification under a mixed prefix—see §5.4.3.)

The need for universal quantification in constraints arises when polymorphism is *required* by the programmer, as opposed to *inferred* by the system. Constraint generation for the universal form is somewhat subtle. A naive definition *fails*:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \# \tau \qquad \textit{Wrong!}$$

This requires τ to be simultaneously equal to *all* of the types that a assumes when $\bar{\alpha}$ varies. For instance, with this incorrect definition, one would have:

$$\begin{array}{ll} & \langle\!\langle \forall\alpha.(\lambda x.x : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ = & \forall\alpha.\langle\!\langle (\lambda x.x : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \equiv & \forall\alpha.(\langle\!\langle \lambda x.x : \alpha \to \alpha \rangle\!\rangle \wedge \alpha = \mathsf{int}) \quad \equiv \quad \forall\alpha.(\mathsf{true} \wedge \alpha = \mathsf{int}) \quad \equiv \quad \mathsf{false} \end{array}$$

A correct definition is:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\exists\gamma.\langle\!\langle a : \gamma \rangle\!\rangle \wedge \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle$$

This requires a to be well-typed *for all* instances of $\bar{\alpha}$ and requires τ to be a valid type for a under *some* instance of $\bar{\alpha}$.

However, a problem with this definition is that the term a is duplicated, which can lead to exponential complexity. Fortunately, this can be avoided modulo a slight extension of the constraint language (Pottier and Rémy, 2003, p. 112). The solution defines:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \mathsf{let}\ x : \forall\vec{\alpha}, \beta[\langle\!\langle a : \beta \rangle\!\rangle].\,\beta\ \mathsf{in}\ x \leq \tau$$

where the new constrain form satisfies the equivalence:

$$\mathsf{let}\ x : \forall\vec{\alpha}, \vec{\beta}[C_1].\,\tau\ \mathsf{in}\ C2 \equiv \forall\vec{\alpha}.\,\exists\vec{\beta}.\,C_1 \wedge \mathsf{def}\ x : \forall\vec{\alpha}, \vec{\beta}[C_1].\,\tau\ \mathsf{in}\ C2$$

Annotating a term with a *type scheme,* rather than just a type, is now just syntactic sugar:

$$(a : \forall\bar{\alpha}.\,\tau) \ \triangleq\ \forall\bar{\alpha}.(a : \tau) \qquad \text{if } \bar{\alpha} \# a$$

In that particular case, constraint generation is in fact simpler:

$$\langle\!\langle (a : \forall\bar{\alpha}.\,\tau) : \tau' \rangle\!\rangle \ \equiv\ \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \wedge (\forall\bar{\alpha}.\,\tau) \leq \tau'$$

Exercise 39 *Check this equivalence.* □

Examples Consider the following two examples:

$$\begin{array}{rl} & \langle\!\langle(\exists\alpha.(\lambda x.\,x+1:\alpha\to\alpha)):\mathsf{int}\to\mathsf{int}\rangle\!\rangle \\ \equiv & \exists\alpha.\langle\!\langle(\lambda x.\,x+1:\alpha\to\alpha):\mathsf{int}\to\mathsf{int}\rangle\!\rangle \\ \equiv & \exists\alpha.(\alpha=\mathsf{int}) \\ \equiv & \mathsf{true} \end{array}$$

$$\begin{array}{rl} & \langle\!\langle(\forall\alpha.(\lambda x.\,x+1:\alpha\to\alpha)):\mathsf{int}\to\mathsf{int}\rangle\!\rangle \\ \Vdash & \forall\alpha.\exists\gamma.\langle\!\langle(\lambda x.\,x+1:\alpha\to\alpha):\gamma\rangle\!\rangle \\ \equiv & \forall\alpha.\exists\gamma.(\alpha=\mathsf{int}\wedge\alpha\to\alpha=\gamma) \\ \equiv & \forall\alpha.\alpha=\mathsf{int} \\ \equiv & \mathsf{false} \end{array}$$

The left-hand side example is well-typed: The system *infers* that α must be int. Because α is a local type variable, it does not appear in the final constraint. The right-hand side example is ill-typed: The system *checks* that α is used in an abstract way, which is not the case here, since the code implicitly assumes that α is int. By contrast, the following example is well-typed:

$$\begin{array}{rl} & \langle\!\langle(\forall\alpha.(\lambda x.\,x:\alpha\to\alpha)):\mathsf{int}\to\mathsf{int}\rangle\!\rangle \\ = & \forall\alpha.\exists\gamma.\langle\!\langle(\lambda x.\,x:\alpha\to\alpha):\gamma\rangle\!\rangle\wedge\exists\alpha.\langle\!\langle(\lambda x.\,x:\alpha\to\alpha):\mathsf{int}\to\mathsf{int}\rangle\!\rangle \\ \equiv & \forall\alpha.\exists\gamma.\alpha\to\alpha=\gamma\wedge\exists\alpha.\alpha=\mathsf{int} \\ \equiv & \mathsf{true} \end{array}$$

The system *checks* that α is used in an abstract way, which is indeed the case here. It also checks that, if α is appropriately instantiated, the code admits the expected type $\mathsf{int}\to\mathsf{int}$.

The two next examples are similar and show the importance of the scope of existential variables. In the first one, the variable α is bound *outside* the let construct;

$$\begin{array}{rl} & \langle\!\langle\exists\alpha.(\mathsf{let}\ f=(\lambda x.\,x:\alpha\to\alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true})):\gamma\rangle\!\rangle \\ \equiv & \exists\alpha.(\mathsf{let}\ f:\alpha\to\alpha\ \mathsf{in}\ \exists\gamma_1\gamma_2.(f\preceq\mathsf{int}\to\gamma_1\wedge f\preceq\mathsf{bool}\to\gamma_2\wedge\gamma_1\times\gamma_2=\gamma)) \\ \equiv & \exists\alpha\gamma_1\gamma_2.(\alpha\to\alpha=\mathsf{int}\to\gamma_1\wedge\alpha\to\alpha=\mathsf{bool}\to\gamma_2\wedge\gamma_1\times\gamma_2=\gamma) \\ \Vdash & \exists\alpha.(\alpha=\mathsf{int}\wedge\alpha=\mathsf{bool}) \\ \equiv & \mathsf{false} \end{array}$$

Then f receives the monotype $\alpha\to\alpha$ and the example is ill-typed. In the other example, α is bound *within* the let construct:

$$\begin{array}{rl} & \langle\!\langle\mathsf{let}\ f=\exists\alpha.(\lambda x.\,x:\alpha\to\alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true}):\gamma\rangle\!\rangle \\ \equiv & \mathsf{let}\ f:\forall\beta[\exists\alpha.(\alpha\to\alpha=\beta)].\,\beta\ \mathsf{in}\ \exists\gamma_1\gamma_2.(f\preceq\mathsf{int}\to\gamma_1\wedge f\preceq\mathsf{bool}\to\gamma_2\wedge\gamma_1\times\gamma_2=\gamma) \\ \equiv & \mathsf{let}\ f:\forall\alpha.\,\alpha\to\alpha\ \mathsf{in}\ \exists\gamma_1\gamma_2.(\ldots) \\ \equiv & \exists\gamma_1\gamma_2.(\mathsf{int}=\gamma_1\wedge\mathsf{bool}=\gamma_2\wedge\gamma_1\times\gamma_2=\gamma) \\ \equiv & \mathsf{int}\times\mathsf{bool}=\gamma \end{array}$$

Here, the term $\exists\alpha.(\lambda x.\,x:\alpha\to\alpha)$ has the same principal type scheme as $\lambda x.\,x$, namely $\forall\alpha.\,\alpha\to\alpha$, which is the type scheme that f receives.

Type annotations in the real world For historical reasons, type variables are not explicitly bound in OCaml. (Retrospectively, that's *bad!*) They are implicitly *existentially* bound at the nearest enclosing toplevel let construct. In Standard ML, type variables are implicitly *universally* bound at the nearest enclosing toplevel let construct. In Glasgow Haskell, type variables are implicitly existentially bound within patterns: *'A pattern type signature brings into scope any type variables free in the signature that are not already in scope'* Peyton Jones and Shields (2004). Constraints help understand these varied design choices uniformly.

5.4.2 Polymorphic recursion

Recall below the typing rule FixAbs for recursive functions, which leads to the derived typing LetRec for recursive definitions:

$$\textsc{FixAbs}\quad \frac{\Gamma, f : \tau \vdash \lambda x.\, a : \tau}{\Gamma \vdash \mu f.\lambda x.a : \tau} \qquad \textsc{LetRec}\quad \frac{\Gamma, f : \tau_1 \vdash \lambda x.\, a_1 : \tau_1 \qquad \bar\alpha \# \Gamma, a_1 \qquad \Gamma, f : \forall \bar\alpha.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let\ rec}\ f\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}$$

These rules require occurrences of f to have *monomorphic type* within the recursive definition (that is, within $\lambda x.\, a_1$). This is visible also in terms of type inference, as the two following constraints are equivalent:

$$\langle\!\langle \mathsf{let\ rec}\ f\ x = a_1\ \mathsf{in}\ a_2 : \tau \rangle\!\rangle \quad \equiv \quad \mathsf{let}\ f : \forall \alpha\beta[\mathsf{let}\ f : \alpha \to \beta; x : \alpha\ \mathsf{in}\ \langle\!\langle a_1 : \beta \rangle\!\rangle].\, \alpha \to \beta\ \mathsf{in}\ \langle\!\langle a_2 : \tau \rangle\!\rangle$$

On the right-hand side, all occurrences of f within a_1 have the same type $\alpha \to \beta$. This is problematic in some situations, most particularly when defining functions over *nested algebraic data types* (Bird and Meertens, 1998; Okasaki, 1999).

This problem is solved by introducing *polymorphic recursion,* that is, by allowing μ-bound variables to receive a polymorphic type scheme, using the following typing rules:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu f.\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ f\ x = a_1\ \mathsf{in}\ a_2 : \tau}$$

This extension of ML is due to Mycroft (1984).

In System F, there is no problem to begin with; no extension is necessary. Polymorphic recursion alters, to some extent, Damas and Milner's type system. Now, not only *let-bound*, but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus. This has two noticeable consequences: *monomorphization,* a technique employed in some ML compilers Tolmach and Oliva (1998); Cejtin et al. (2007), is no longer possible; besides, *type inference* becomes problematic!

Type inference for ML with polymorphic recursion is undecidable Henglein (1993). It is equivalent to the undecidable problem of *semi-unification.* Yet, type inference in the presence of polymorphic recursion can be made simple by relying on a *mandatory type annotation.*

The syntax and typing rules for recursive definitions become:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu(f : \sigma).\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ (f : \sigma) = \lambda x.\, a_1\ \mathsf{in}\ a_2 : \tau}$$

The type scheme σ no longer has to be guessed. With this feature, contrary to what was said earlier (p. 109), *type annotations are not just restrictive:* they are sometimes required for type inference to succeed. The constraint generation rule becomes:

$$\langle\!\langle \mathsf{let\ rec}\ (f : \sigma) = \lambda x.\, a_1\ \mathsf{in}\ a_2 : \tau \rangle\!\rangle \quad = \quad \mathsf{let}\ f : \sigma\ \mathsf{in}\ (\langle\!\langle \lambda x.\, a_1 : \sigma \rangle\!\rangle \wedge \langle\!\langle a_2 : \tau \rangle\!\rangle)$$

It is clear that f receives type scheme σ both *inside and outside* of the recursive definition.

5.4.3 Unification under a mixed prefix

Unification under a mixed prefix means unification in the presence of both existential and universal quantifiers. We extend the basic unification algorithm with support for universal quantification. The solved forms are unchanged: universal quantifiers are always *eliminated.*

In short, in order to reduce $\forall\bar{\alpha}.C$ to a solved form, where C is itself a solved form—see (Pottier and Rémy, 2003, p. 109) for details:

- If a rigid variable is equated with a constructed type, fail.
  For example, $\forall\alpha.\exists\beta\gamma.(\alpha = \beta \to \gamma)$ is false.

- If two rigid variables are equated, fail.
  For example, $\forall\alpha\beta.(\alpha = \beta)$ is false.

- If a free variable dominates a rigid variable, fail.
  For example, $\forall\alpha.\exists\beta.(\gamma = \alpha \to \beta)$ is false.

- Otherwise, one can decompose C as $\exists\bar{\beta}.(C_1 \wedge C_2)$, where $\bar{\alpha}\bar{\beta} \mathrel{\#} C_1$ and $\exists\bar{\beta}.C_2 \equiv \mathsf{true}$; in that case, $\forall\bar{\alpha}.C$ reduces to just C_1.

  For example, $\forall\alpha.\exists\beta\gamma_1\gamma_2.(\beta = \alpha \to \gamma \wedge \gamma = \gamma_1 \to \gamma_2)$ reduces to just $\exists\gamma_1\gamma_2.(\gamma = \gamma_1 \to \gamma_2)$. The constraint $\forall\alpha.\exists\beta.(\beta = \alpha \to \gamma)$ is equivalent to true.

OCaml implements a form of unification under a mixed prefix. This is illustrated by the following interactive OCaml session:

```
let module M : sig val id : 'a → 'a end = struct let id x = x + 1 end in M.id
```

Values do not match: val id : int → int
is not included in val id : 'a → 'a

This gives rise to a constraint of the form $\forall\alpha.\alpha = \mathsf{int}$, while the following example gives rise to a constraint of the form $\exists\beta.\forall\alpha.\alpha = \beta$:

```
let r = ref (fun x → x) in
let module M : sig val id : 'a → 'a end = struct let id = !r end in M.id;;
```

Values do not match: val id : '_a → '_a
is not included in val id : 'a → 'a

5.5 Equi- and iso-recursive types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees. Indeed, if the grammar of types is $\tau ::= \mathsf{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values. For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not: "A list is either empty or a pair of an element and a list." We need something like this:

$$\mathsf{list}\ \alpha \quad \diamond \quad \mathsf{unit} + \alpha \times \mathsf{list}\ \alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches. In the equi-recursive approach, a recursive type is *equal* to its unfolding. In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions.*

5.5.1 Equi-recursive types

In the equi-recursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid F\ \vec{\tau}$$

is no longer interpreted inductively. Instead, types are the *regular trees* built on top of this signature. If desired, it is possible to use *finite syntax* for recursive types:

$$\tau ::= \alpha \mid \mu\alpha.(F\ \vec{\tau})$$

We do not allow the seemingly more general $\mu\alpha.\tau$, because $\mu\alpha.\alpha$ is meaningless, and $\mu\alpha.\beta$ or $\mu\alpha.\mu\beta.\tau$ are useless. If we write $\mu\alpha.\tau$, it should be understood that τ is *contractive,* that is, τ is a type constructor application. For instance, the type of lists of elements of type α is:

$$\mu\beta.(\mathsf{unit} + \alpha \times \beta)$$

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding.* Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

One can also prove Brandt and Henglein (1998) that equality is the least congruence generated by the following two rules:

$$\textsc{Fold/Unfold}\quad \mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau \qquad\qquad \textsc{Uniqueness}\quad \dfrac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, τ must be contractive. This axiomatization does not directly lead to an efficient algorithm for deciding equality, though. In the presence of equi-recursive types, structural induction on types is no longer permitted—but *we never used it* anyway. It remains true that $F\,\vec{\tau}_1 = F\,\vec{\tau}_2$ implies $\vec{\tau}_1 = \vec{\tau}_2$—this was used in our Subject Reduction proofs. It remains true that $F_1\,\vec{\tau}_1 = F_2\,\vec{\tau}_2$ implies $F_1 = F_2$—this was used in our Progress proofs. So, the reasoning that leads to *type soundness* is unaffected.

Exercise 40 *Prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from* `Inductive` *to* `CoInductive`*.* □

How is type inference adapted for equi-recursive types? The *syntax* of constraints is unchanged: they remain systems of equations between finite first-order types, without μ's. Their *interpretation* changes: they are now interpreted in a universe of regular trees. As a result, constraint generation is *unchanged*; constraint solving is adapted by *removing the occurs check*.

Exercise 41 *Describe solved forms and show that every solved form is either* false *or satisfiable.* □

Here is a function that measures the length of a list:

$$\mu(\mathit{length}).\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \lambda().\,0 \diamond \lambda(y, z).\,1 + \mathit{length}\ z$$

Type inference gives rise to the *cyclic equation* $\beta = \mathsf{unit} + \alpha \times \beta$, where *length* has type $\beta \rightarrow \mathsf{int}$. That is, *length* has *principal type scheme:* $\forall\alpha.\,(\mu\beta.\mathsf{unit}+\alpha\times\beta) \rightarrow \mathsf{int}$ or, equivalently, principal constrained type scheme: $\forall\alpha[\beta = \mathsf{unit}+\alpha\times\beta].\,\beta \rightarrow \mathsf{int}$. The cyclic equation that characterizes lists was never provided by the programmer, but was inferred.

OCaml implements equi-recursive types upon explicit request, launching the interactive session with the command "`ocaml -rectypes`":

```
type ('a, 'b) sum = Left of 'a | Right of 'b
  type ('a, 'b) sum = Left of 'a | Right of 'b

let rec length x = function Left () → 0 | Right (y, z) → 1 + length z
val length : ((unit, 'b * 'a) sum as 'a) → int = <fun>
```

Notice that `-rectypes` is only an option which is not on by default. Equi-recursive types are simple and powerful, but in practice, they are perhaps *too expressive*. Continuing with in the `-rectype` option:

$\texttt{let rec}\ map\ f = \texttt{function}\ [\,] \to [\,] \mid y :: z \to map\ f\ y :: map\ f\ z$

val map : 'a → ('b list as 'b) → ('c list as 'c) = ⟨fun⟩

$map\ (\texttt{fun}\ x \to x + 1)\ [\ 1;\ 2\ ]$

This expression has type int but is used with type 'a list as 'a

$map\ ()\ [[\,];\ \ [[\,]]]$

– : 'a list as 'a = [[]; [[]]]

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error. Hence, by default, OCaml typechecker reject type cycles that do not involve an object type or a variant type. In a normal OCaml session (no `-rectypes`), the following is still accepted, though:

$\texttt{let}\ f\ x = x\#hello\ x;;$

val f : (< hello : 'a → 'b; .. > as 'a) → 'b = ⟨fun⟩

OCaml implements a partial occurs check that stops at object and variant types: equi-recursive types are allowed provided every infinite path crosses an object or a variant type.

5.5.2 Iso-recursive types

In the iso-recursive approach, the user is allowed to introduce new *type constructors* D via (possibly mutually recursive) *declarations:*

$$D\,\vec{\alpha} \approx \tau \qquad (\text{where } \mathsf{ftv}(\tau) \subseteq \bar{\alpha})$$

Each such declaration adds a unary constructor fold_D and a unary destructor unfold_D with the following types and the new reduction rule:

$$\mathsf{fold}_D : \forall \bar{\alpha}. \tau \to D\,\vec{\alpha} \qquad \mathsf{unfold}_D : \forall \bar{\alpha}. D\,\vec{\alpha} \to \tau \qquad \mathsf{unfold}_D\,(\mathsf{fold}_D\ v) \longrightarrow v$$

Ideally, iso-recursive types should not have any runtime cost. One solution is to compile constructors and destructors away into a target language with equi-recursive types. Another solution is to see iso-recursive types as a restriction of equi-recursive types where the source language does not have equi-recursive types but instead two unary destructors fold_D and unfold_D with the semantics of the identity function. Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the UNIQUENESS typing rule.

For, example iso-recursive lists can be defined as follows. A parametrized, iso-recursive type of lists is: $\mathsf{list}\ \alpha \approx \mathsf{unit} + \alpha \times \mathsf{list}\ \alpha$. The empty list is: $\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ())\ :\ \forall\alpha.\, \mathsf{list}\ \alpha$. A function that measures the length of a list is:

$$\mu(\mathit{length}).\lambda xs.\mathsf{case}\ (\mathsf{unfold}_{\mathsf{list}}\ xs)\ \mathsf{of}\ \lambda().\, 0 \diamond \lambda(x, xs).\, 1 + \mathit{length}\, xs \quad : \quad \forall\alpha.\, \mathsf{list}\ \alpha \to \mathsf{int}$$

One *folds upon construction* and *unfolds upon deconstruction.*

In the iso-recursive approach, *types remain finite.* The type $\mathsf{list}\ \alpha$ is just an application of a type constructor to a type variable. As a result, *type inference is unaffected.* The occurs check remains.

5.5.3 Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums. This suppresses the *verbosity* of explicit folds and unfolds as well as the *fragility* and inconvenience of numeric indices—instead, named *record fields* and *data constructors* are used. For instance,

$$\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ()) \quad \text{is replaced with} \quad \mathit{Nil}\,()$$

An algebraic data type constructor D is introduced via a *record type* or *variant type* definition:

$$D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell \qquad\qquad \text{or} \qquad\qquad D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$$

The set L denotes a finite set of record labels or data constructors $\{\ell_1 \ldots \ell_n\}$, which is fixed for a given definition. Algebraic data type definitions can be mutually recursive.

The record type definition $D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell$ introduces a record n-ary *constructor* and n record unary *destructors* with the following types:

$$C ::= \ldots \mid \{\ell_1 = \cdot\,, \ldots \ell_n = \cdot\,\} \qquad\qquad d ::= \ldots \mid (\,\cdot\,.\ell_1) \mid \ldots (\,\cdot\,.\ell_n)$$

$$\{\ell_1 = \cdot\,, \ldots \ell_n = \cdot\,\} : \forall\vec{\alpha}.\, \tau_{\ell_1} \to \ldots \tau_{\ell_n} \to D\,\vec{\alpha} \qquad\qquad \cdot\,.\ell : \forall\vec{\alpha}.\, D\,\vec{\alpha} \to \tau_\ell$$

The variant type definition $D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$ introduces unary variant constructors and variant destructor of arity $n+1$ with the following types:

$$C ::= \ldots \mid (\ell\,\cdot) \qquad d ::= \ldots \mid \mathsf{case}\ \cdot\ \mathsf{of}\ [\ell_1 : \cdot \diamond \ldots \ell_n : \cdot] \qquad \ell : \forall\vec{\alpha}.\, \tau_\ell \to D\,\vec{\alpha}$$

$$\mathsf{case}\ \cdot\ \mathsf{of}\ [\ell_1 : \cdot \diamond \ldots \ell_n : \cdot] : \forall\vec{\alpha}\beta.\, D\,\vec{\alpha} \to (\tau_{\ell_1} \to \beta) \to \ldots (\tau_{\ell_n} \to \beta) \to \beta$$

For example, an algebraic data type of lists is $\mathsf{list}\ \alpha \approx \mathit{Nil} : \mathsf{unit} + \mathit{Cons} : \alpha \times \mathsf{list}\ \alpha$ gives rise to:

$$\begin{aligned}
\mathsf{case}\ \cdot\ \mathsf{of}\ [\mathit{Nil} : \cdot \diamond \ldots \mathit{Cons} : \cdot] :&\quad \forall\alpha\beta.\, \mathsf{list}\ \alpha \to (\mathsf{unit} \to \beta) \to ((\alpha \times \mathsf{list}\ \alpha) \to \beta) \to \beta \\
\mathit{Nil} :&\quad \forall\alpha.\, \mathsf{unit} \to \mathsf{list}\ \alpha \\
\mathit{Cons} :&\quad \forall\alpha.\, (\alpha \times \mathsf{list}\ \alpha) \to \mathsf{list}\ \alpha
\end{aligned}$$

$$\frac{\sigma = \Gamma(x) \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma}\ \textsc{hm-Var} \qquad \frac{C, (\Gamma, x : \tau_0) \vdash a : \tau}{C, \Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{hm-Abs} \qquad \frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{hm-App}$$

$$\frac{C, \Gamma \vdash a_1 : \sigma \qquad C, (\Gamma, x : \sigma) \vdash a_2 : \tau}{C, \Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau}\ \textsc{hm-Let} \qquad \frac{C \wedge C_0, \Gamma \vdash a : \tau \qquad \vec\alpha \# C, \Gamma}{C \wedge \exists\vec\alpha. C_0, \Gamma \vdash a : \forall\vec\alpha[C_0].\, \tau}\ \textsc{hm-Gen}$$

$$\frac{C, \Gamma \vdash a : \forall\vec\alpha[C_0].\, \tau}{C \wedge C_0, \Gamma \vdash a : \tau}\ \textsc{hm-Inst} \qquad \frac{C, \Gamma \vdash a : \tau_1 \qquad C \Vdash \tau_1 \le \tau_2}{C, \Gamma \vdash a : \tau_2}\ \textsc{hm-Sub} \qquad \frac{C, \Gamma \vdash a : \tau \qquad \vec\alpha \# \Gamma, \tau}{\exists\vec\alpha. C, \Gamma \vdash a : \tau}\ \textsc{hm-Exists}$$

Figure 5.7: Typing rules for HM(X)

A function that measures the length of a list is:

$$\mu(\mathit{length}).\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathit{Nil} : \lambda().\, 0 \diamond \mathit{Cons} : \lambda(y,\, z).\, 1 + \mathit{length}\ z \quad : \quad \forall\alpha.\, \mathsf{list}\ \alpha \to \mathsf{int}$$

Mutable record fields In OCaml, a record field can be marked *mutable*. This introduces an extra binary destructor for writing this field: $(\,\cdot\,.\ell \leftarrow \cdot)$ of type $\forall\vec\alpha.\, D\,\vec\tau \to \tau_\ell \to \mathsf{unit}$. However, this also makes record construction a destructor since, when fully applied it is *not a value* but it allocates a piece of store and returns its location. Thus, due to the value restriction, the type of such expressions cannot be generalized.

5.6 HM(X)

Soundness and completeness of type inference are in fact easier to prove if one adopts a *constraint-based specification* of the type system, as in the language HM(X) introduced by Odersky et al. (1999).

In HM(X), judgments take the form $C, \Gamma \vdash a : \tau$, called a constrained typing judgments. Read *under the assumption C and typing environment Γ, the program a has type τ.* Here C constrains free type variables of the judgment while Γ provides the type of free program variables of a. The constraint C ranges over first-order typing constraints—except that we require type constraints to have no free program variables. In a constrained typing judgment $C, \Gamma \vdash a : \tau$,

The parameter X in HM(X) stands for the logic of the constraint language. We have so far only consider constraints with an equality predicate. However, the equality replaced may be by an asymmetric subtyping predicate $\le$, which makes the language of constraints richer.

The typing rules also use an entailment predicate $C \Vdash C'$ between constraints that is more general than constraint equivalence. Entailment is defined as expected: $C \Vdash C'$ if and only if any ground assignment that satisfies C also satisfies C'.

Typing rules for $\mathsf{HM(X)}$ are presented in Figure 5.7. Moreover, judgment are taken up to constraint equivalence. The constraint $\exists\sigma$ in the premise of Rule HM-VAR is an abbreviation for $\exists\bar{\alpha}.C_0$ where σ is $\forall\bar{\alpha}[C_0].\tau$. A valid judgment is one that has a derivation with those typing rules. In a valid judgment, C may not be satisfiable. A program is well-typed in environment Γ if it has a valid judgment $C, \Gamma \vdash a : \tau$ for some τ and *satisfiable* constraint C.

When considering equality only constraints, $\mathsf{HM}(=)$ is in fact equivalent to ML: if Γ and τ contain only Damas-Milner's type schemes, then $\Gamma \vdash a : \tau$ in ML if and only if $\mathsf{true}, \Gamma \vdash a : \tau$ in $\mathsf{HM(X)}$. Moreover, if $C, \Gamma \vdash a : \tau$ in $\mathsf{HM(X)}$ and φ is an idempotent solution of C, we have $\mathsf{true}, \Gamma_\varphi \vdash a : \tau_\varphi$ in $\mathsf{HM(X)}$ where $(\cdot)_\varphi$ translates $\mathsf{HM(X)}$ type schemes into ML type schemes—applying the substitution φ on the fly.

As for ML, there is an equivalent syntax-directed presentation of the typing rules. However, we may take advantage of program variables in constraints to go one step further and mix the constraint C (without free program variables) and the typing environment Γ into a single constraint C now with possibly free program variables. Judgments take the form $C \vdash a : \tau$ where C constrains type variables and assign constrained type schemes to program variables. The type system, called $\mathsf{PCB(X)}$, is described on Figure 5.8. It is equivalent to $\mathsf{HM(X)}$—see (Pottier and Rémy, 2005) for the precise comparison.

For example of a derivation in $\mathsf{PCB(X)}$, let a be $\mathsf{let}\; y = \lambda x.\, x \;\mathsf{in}\; y\; y$:

$$
\textsc{Exists}\;\dfrac{\textsc{Let}\;\dfrac{\textsc{Fun}\;\dfrac{\overset{\textsc{Var}}{x \le \alpha \vdash x : \alpha}}{\mathsf{let}\; x : \alpha_0 \;\mathsf{in}\; x \le \alpha \vdash \lambda x.\, x : \alpha_0 \to \alpha} \qquad \textsc{App}\;\dfrac{\overset{\textsc{Var}}{y \le \beta_2 \to \beta_1 \vdash y : \beta_2 \to \beta_1} \qquad \overset{\textsc{Var}}{y \le \beta_2 \vdash y : \beta_2}}{y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \vdash y\; y : \beta_1}}{C \vdash a : \beta_1}}{\exists\beta_2.C \vdash a : \beta_1}
$$

where C is

$$
\mathsf{let}\; y : \forall\alpha\alpha_0[\mathsf{let}\; x : \alpha_0 \;\mathsf{in}\; x \le \alpha].\, \alpha_0 \to \alpha \;\mathsf{in}\; y \le \beta_2 \to \beta_1 \wedge y \le \beta_2
$$

The constraint C can be simplified as follows:

$$
\begin{array}{rcl}
\exists\beta_2.C & = & \exists\beta_2.\; \mathsf{let}\; y : \forall\alpha\alpha_0[\alpha_0 = \alpha].\, \alpha_0 \to \alpha \;\mathsf{in}\; y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \\
& \equiv & \exists\beta_2.\; \mathsf{let}\; y : \forall\alpha.\, \alpha \to \alpha \;\mathsf{in}\; y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \\
& \equiv & \exists\beta_2\alpha_1\alpha_2.\;\; \alpha_1 \to \alpha_1 = \beta_2 \to \beta_1 \;\;\wedge\;\; \alpha_2 \to \alpha_2 = \beta_2 \\
& \equiv & \exists\alpha.\; \beta_1 = \alpha \to \alpha
\end{array}
$$

Hence, we also have $\exists\alpha.\; \beta_1 = \alpha \to \alpha \vdash a : \beta_1$. This is a valid judgment, but not a satisfiable one. However, by rule PCB-SUB and PCB-EXISTS, we have $\exists\beta_1.\, (\exists\alpha.\; \beta_1\alpha \to \alpha) \wedge \beta_1 = \beta \to \beta) \vdash a : \beta \to \beta$, which is equivalent to $\mathsf{true} \vdash a : \beta \to \beta$ and is both valid and satisfiable.

The type inference algorithm for ML is sound and complete for $\mathsf{PCB(X)}$:

– *Soundness:* $\langle\!\langle a : \tau \rangle\!\rangle \vdash a : \tau$. The constraint inferred for a typing validates the typing.

– *Completeness:* If $C \vdash a : \tau$ then $C \Vdash \langle\!\langle a : \tau \rangle\!\rangle$. The constraint inferred for a typing is more general than any constraint that validates the typing.

$$\frac{C \Vdash x \le \tau}{C \vdash x : \tau}\ \textsc{pcb-Var} \qquad \frac{C \vdash a : \tau}{\mathsf{let}\ x : \tau_0\ \mathsf{in}\ C \vdash a : \tau_0 \to \tau}\ \textsc{pcb-Abs} \qquad \frac{C_1 \vdash a_1 : \tau_2 \to \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{C_1 \wedge C_2 \vdash a_1\ a_2 : \tau_1}\ \textsc{pcb-App}$$

$$\frac{C_1 \vdash a_1 : \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{\mathsf{let}\ x : \forall \mathcal{V}[C_1].\,\tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}\ \textsc{pcb-Let} \qquad \frac{C \vdash a : \tau_1}{C \wedge \tau_1 \le \tau_2 \vdash a : \tau_2}\ \textsc{pcb-Sub} \qquad \frac{C \vdash a : \tau \qquad \alpha \# \tau}{\exists \alpha.\, C \vdash a : \tau}\ \textsc{pcb-Exists}$$

Figure 5.8: Typing rules for PCB(X)

Note Our presentation of HM(X) is incomplete. See also Skalka and Pottier (2002) for a more recent presentation of HM(X) and Pottier and Rémy (2005) for a detailed presentation of several variants of HM(X).

Our proof of type soundness for ML only applies for HM(=). One may prove type soundness for HM(X) in the general case for some logic X, under the axiom that the arrow type constructor is contra-variant for subtyping. See Pottier and Rémy (2005).

5.7 Type reconstruction in System F

Type checking in explicitly-typed System F is easy. Still, an implementation must carefully deal with variable bindings and renaming when applying type substitutions. However, as we have seen, programming with fully-explicit types is unpractical.

Full type inference in System F has long been an open problem, until Wells (1999) proved it undecidable by showing that it is equivalent to the semi-unification problem which was earlier proved undecidable. (Notice that the full type-inference problem is not directly related to second-order unification but rather to semi-unification.)

Hence, we must perform *partial type inference* in System F. Either type inference is incomplete, or some amount of type annotations must be provided. Several solutions are used in practice. They alleviate the need for a lot of redundant type annotations.

5.7.1 Type inference based on Second-order unification

Full type inference is equivalent to semi-unification. However, type inference becomes equivalent to second-order unification if all the positions of type abstractions and type applications are explicit, while types are themselves left implicit. That is, if terms are

$$M ::= x \mid \lambda x{:}?.\,M \mid M\ M \mid \Lambda ?.M \mid M\ ?$$

where the question marks stand for type variables and types to be inferred. Although, the problem of second-order unification is undecidable, there are semi-algorithms that often work well in common cases. This method was proposed by Pfenning (1988).

$$\frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau}\;\textsc{Var-I} \qquad \frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau}\;\textsc{Abs-C} \qquad \frac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\, a_2 \Uparrow \tau_1}\;\textsc{App-I} \qquad \frac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau}\;\textsc{I-C}$$

$$\frac{\Gamma \vdash a \Downarrow \tau}{\Gamma \vdash (a : \tau) \Uparrow \tau}\;\textsc{Annot-I} \qquad \frac{\Gamma, x : \tau_0 \vdash a \Uparrow \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, a \Uparrow \tau_0 \to \tau}\;\textsc{Abs-I}$$

Figure 5.9: Bidirectional type checking for the simply-typed λ-calculus .

In fact, partial type inference based on second-order unification can be mixed with type checking. Explicit polymorphism may be reintroduced as in explicitly-typed System F while explicitly-controlled implicit instantiation can be performed as above by second-order unification. The source language is:

$$M ::= x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid \lambda x{:}?.\, M \mid M\ ? \mid \mathsf{let}\ f = \Lambda^?\alpha_1.\ldots\Lambda^?\alpha_n.M\ \mathsf{in}\ M$$

The new let-binding form is used to declare type arguments that will be made implicit. Then, every occurrence of such a variable automatically adds type-application holes at the corresponding positions and type parameters will be inferred using second-order unification. This amounts to understanding the new let-binding form as follows:

$$\mathsf{let}\ f = \Lambda^?\alpha_1.\ldots\Lambda^?\alpha_n.M_1\ \mathsf{in}\ M_2 \ \triangleq\ \mathsf{let}\ f = \Lambda\alpha_1.\ldots\Lambda\alpha_n.M_1\ \mathsf{in}\ [f \mapsto f\ \ ?\ldots\ ?]M_2$$

Type inference in this language still reduces to second-order unification.

5.7.2 Bidirectional type inference

Type-checking in explicit simply-typed λ-calculus is easy because typing rules have an algorithmic reading. This implies that they are syntax directed, but also that judgments can be read as functions where some arguments are inputs and others are output. In the implicit calculus, the rules are still syntax-directed, but some of them do not have an obvious algorithmic reading. Typically, Γ and a would be inputs and τ is an output in the judgment $\Gamma \vdash a : \tau$, which we may represent as $\Gamma^\uparrow \vdash a^\uparrow : \tau^\downarrow$. However, in the rule for abstraction:

$$\frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\;\textsc{Abs}$$

the type τ_0 is used both as input (in the premise) and as an output in the conclusion. Hence, type-checking the implicit simply-typed λ-calculus is not straightforward. In some cases, the type of the function may be known, *e.g.* when the function is an argument to an expression of a known type. Then, it suffices to check the proposed type is indeed correct.

Formally, we need algorithmic reading of the typing judgment, depending on whether the return type is known or unknown. We may split the typing judgment $\Gamma \vdash a : \tau$ into two

$$
\text{Abs-C}\;\dfrac{\text{I-C}\;\dfrac{\text{App-I}\;\dfrac{\text{Var-I}\;\dfrac{}{\Gamma \vdash f \Uparrow \tau} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau_1 \vdash x \Uparrow \tau_1}\;\text{Var-I}}{\Gamma, x : \tau_1 \vdash x \Downarrow \tau_1}\;\text{C-I}}{\Gamma \vdash \lambda x.\, x \Downarrow \tau_1 \to \tau_1}\;\text{Abs-C}}{\Gamma \vdash f\ (\lambda x.\, x) \Uparrow \tau_2}}{\Gamma \vdash f\ (\lambda x.\, x) \Downarrow \tau_2}}{\varnothing \vdash \lambda f{:}\tau.\, f\ (\lambda x.\, x) \Downarrow \tau \to \tau_2}
$$

Figure 5.10: Example of bidirectional derivation

judgments $\Gamma \vdash a \Downarrow \tau$ to check that a may be assigned the type τ and $\Gamma \vdash a \Uparrow \tau$ to infer the type τ of a (or with information flows $\Gamma^{\uparrow} \vdash a^{\uparrow} \Downarrow \tau^{\uparrow}$ and $\Gamma^{\uparrow} \vdash a^{\uparrow} \Uparrow \tau^{\downarrow}$. Both judgments are recursively defined by the rules of Figure **??**: the checking mode can call the inference mode when needed; conversely, annotations may be used to turn inference mode into checking mode. (As a particular case, annotations on type abstractions enable the inference mode.)

An example of bidirectional derivation is given on Figure 5.10. The type τ stands for $(\tau_1 \to \tau_1) \to \tau_2$ and the environment Γ is $f : \tau$.

The bidirectional method can be extended to deal with polymorphic types, but it is more complicated. The idea, due to Cardelli (1993), was popularized by Pierce and Turner (2000), and Odersky et al. (2001) and is still being improved Dunfield (2009).

Predicative polymorphism *Predicative polymorphism* is an interesting subcase of bidirectional type inference in the presence of predicative polymorphism. Predicative polymorphism is a restriction of impredicative polymorphism as can be found in System F. With predicative polymorphism, types are stratified so that polymorphic types can only be instantiated with simple types.

Interestingly, partial type inference can then still reduced to typing constraints under a mixed prefix (Rémy, 2005; Jones et al., 2006). Unfortunately, predicative polymorphism is too restrictive for use in programming languages: as polymorphic values often need to be put in data-structures whose constructors are polymorphic but impredicative polymorphism does not allow implicit instantiation of polymorphic constructors by polymorphic types.

One may also use a hierarchy of types where polymorphic types of rank n can be instantiated with polymorphic types of a strictly lower rank. This increases expressiveness but F is still more expressive than the union of all F^n.

Type inference with first-order constraints does not work for higher ranks.

Local type inference A simpler approach than *global* bidirectional type inference proposed by Pierce and Turner and improved by Odersky et al. is to perform bidirectional type inference *locally*, *i.e.* by considering for each node only a small context surrounding it.

Subtyping Interestingly, bidirectional type inference can easily be extended to work in the presence of subtyping, which is not the case for methods based on second order unification.

5.7.3 Partial type inference in MLF

The language MLF (Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008) is an extension of System F especially designed for partial type inference—in fact for type inference a la ML within System F. That is, the inference algorithm performs first-order unification and aggressive ML-style let-generalization, but in the presence of second-order types. Interestingly, only parameters of functions that are used polymorphically need to be annotated in MLF; type abstractions and type annotation are always left implicit. However, for the purpose of type inference, MLF introduces richer types that enable to write "more principal types", but that are also harder to read. The type inference method for MLF can be seen as a generalization of the constraint-based type inference for ML that handles polymorphic types.

5.8 Proofs and Solution to Exercises

Proof of Theorem 15

We prove $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ if and only if $\phi\Gamma \vdash a : \phi\tau$ by induction on a. We prove both implications independently because reasoning with equivalence is error-prone, since the arguments are similar but often not quite the same in both directions. The proof is thus a bit lengthy, but all cases are easy.

Case a is x: Assume $\phi\Gamma \vdash a : \phi\tau$. By inversion of typing, this judgment must be derived by rule Var. Hence, $\phi\tau = \phi\Gamma(x)$. By definition of satisfiability this implies $\phi \vdash \tau = \Gamma(x)$. By definition of typing constraint, this is $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

Conversely, assume $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$. By definition of typing constraint, this is $\phi \vdash \tau = \Gamma(x)$. By inversion of satisfiability we must have $\phi\tau = \phi\Gamma(x)$. Hence, by rule Var, we have $\phi\Gamma \vdash a : \phi\tau$.

Case a is $a_1\ a_2$: Assume $\phi\Gamma \vdash a : \phi\tau$. By rule App, there exists τ_2 such that $\phi\Gamma \vdash a_1 : \tau_2 \to \phi\tau$ and $\phi\Gamma \vdash a_2 : \tau_2$. Let $\beta \# \Gamma$ and ϕ' be $\phi, \beta \mapsto \tau_2$. We have $\phi'\Gamma \vdash a_1 : \phi'\beta \to \tau$ and $\phi'\Gamma \vdash a_2 : \beta$. Hence, by induction hypothesis $\phi' \vdash \langle\!\langle \Gamma \vdash a_1 : \beta \to \tau \rangle\!\rangle$ and $\phi' \vdash \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle$. Thus, $\phi \vdash \exists\beta. \langle\!\langle \Gamma \vdash a_1 : \beta \to \tau \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle$. *i.e.* $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

Conversely, assume $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$. We have $\phi \vdash \exists\beta. \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_1 : \beta \to \tau \rangle\!\rangle$. We may assume *w.l.o.g.* that $\beta \# \phi$. There must exist ϕ' of the form $\phi, \beta \mapsto \tau_2$ such that $\phi' \vdash \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_1 : \beta \to \tau \rangle\!\rangle$. By induction hypothesis, this implies $\phi'\Gamma \vdash a_2 : \phi'\beta$ and $\phi'\Gamma \vdash a_1 : \phi'\beta \to \tau$, *i.e.* $\phi\Gamma \vdash a_2 : \tau_2$ and $\phi\Gamma \vdash a_1 : \phi\tau_2 \to \tau$. By rule App, we have $\phi\Gamma \vdash a_1\ a_2 : \phi\tau$.

Case a is $\lambda x.\, a_1$: Assume $\phi\Gamma \vdash a : \phi\tau$. We may assume *w.l.o.g.* that $x \# \Gamma$. By rule Fun, there must exist τ_1 and τ_2 such that $\phi\Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi\tau = \tau_2 \to \tau_1$. Let β_1 and β_2 be disjoint from Γ and ϕ' be $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$. Then, both $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1$ and $\phi'\tau = \phi'(\beta_2 \to \beta_1)$ hold. By induction hypothesis, $\phi' \vdash \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \tau_1 \rangle\!\rangle$ and $\phi' \vdash \tau = \beta_2 \to \beta_1$ Therefore, $\phi \vdash \exists \beta_1 \beta_2. \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle\!\rangle \wedge \tau = \beta_2 \to \beta_1$. That is, $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

Conversely, assume $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$. By definition of constraints, we have $\phi \vdash \exists \beta_1 \beta_2. \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle\!\rangle \wedge \tau = \beta_2 \to \beta_1$ for some x disjoint from Γ. We may assume *w.l.o.g.* that $\beta_1, \beta_2 \# \phi$. There must exist ϕ' of the form $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$ such that $\phi' \vdash \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \tau_1 \rangle\!\rangle$ and $\phi' \vdash \tau = \beta_2 \to \beta_1$. By induction hypothesis, $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1$ and $\phi'\tau = \phi'(\beta_2 \to \beta_1)$. That is, $\phi\Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi\tau = \tau_2 \to \tau_1$. Hence, by rule Fun, we have $\phi\Gamma \vdash a : \phi\tau$.

Solution of Exercise 36

See Bjørner (1994). ∎

Solution of Exercise 37

Consider the module $\mathsf{struct}\ f = \mathsf{let}\ f = \lambda x. x\ \mathsf{in}\ f\ f\ \mathsf{end}$. In core ML, the expression has principal type $\alpha \to \alpha$—but α cannot be generalized. Hence, $\mathsf{sig}\ f : \forall \alpha. \alpha \to \alpha\ \mathsf{end}$ is not a signature for this module; nor is $\mathsf{sig}\ f : \alpha \to \alpha\ \mathsf{end}$ since it is not a well-formed one. Correct signatures are $\mathsf{sig}\ f : \tau \to \tau\ \mathsf{end}$ for any τ, but they do not have a best element. ∎

Chapter 6

Existential types

Compilation is type-preserving when each intermediate language is *explicitly typed*, and each compilation phase transforms a typed program into a typed program in the next intermediate language.

Type preserving compilation is interesting for several reasons: it can help debug the compiler; types can be used to drive optimizations; types can also be used to produce *proof-carrying code*; proving that types are preserved during compilation can be the first step towards proving that the *semantics* is preserved Chlipala (2007).

Besides, type-preserving compilation is quite challenging as it exhibits an encoding of programming constructs into programming language that usually requires richer type systems. Sometimes, an encoding later becomes a programming idiom that is used directly in the source language. There are several examples: closure conversion requires an extension of the language with existential types, which happens to very useful on their own. Closures are themselves a simple form of objects. Defunctionalization may be done manually on some particular programs, *e.g.* in web applications to monitor the computation.

A classic paper by Morrisett *et al.* 1999 shows how to go from System F to "Typed Assembly Language", while preserving types along the way. Its main passes are:

1. *CPS conversion* fixes the order of evaluation, names intermediate computations, and makes all function calls tail calls;

2. *closure conversion* makes environments and closures explicit, and produces a program where all functions are closed;

3. allocation and initialization of tuples is made explicit;

4. the calling convention is made explicit, and variables are replaced with (an unbounded number of) machine registers.

In general, a type-preserving compilation phase involves not only a translation of *terms*, mapping M to $[\![M]\!]$, but also a translation of *types*, mapping τ to $[\![\tau]\!]$, with the property:

$$\Gamma \vdash M : \tau \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$$

The translation of types carries a lot of information: examining it is often enough to guess what the translation of terms will be.

6.1 Towards typed closure conversion

First-class functions may appear in the body of other functions. hence, their own body may contain free variables that will be bound to values during the evaluation in the execution environment. Because they can be returned as values, and thus used outside of their definition environment, they must store their execution environment in their value. A *closure* is the packaging of the code of a first-class function with its runtime environment, so that it becomes closed, *i.e.* independent of the runtime environment and can be passed to another function and applied in another runtime environment. Closures can also be used to represent recursive functions and objects in the object-as-record-of-methods paradigm.

In the following, the *source* calculus has *unary* λ-abstractions, which can have free variables, while the *target* calculus has *binary* λ-abstractions, which must be *closed*. In the target language, we also use pattern matching over tuples. The translation will be naive, insofar as it will not handle functions of multiple arguments in a special way. One could argue that this is a feature, not a limitation, and that "uncurrying" (if desired) should be a separate type-preserving pass anyway. But closure conversion can also be easily extended to n-ary functions.

There are at least two variants of closure conversion: In the *closure-passing variant*, the closure and the environment are a single memory block; In the *environment-passing variant*, the environment is a separate block, to which the closure points. The impact of this choice on the term translations is minor. Closure-passing better supports simple recursive functions; but this is less obvious with mutually recursive ones. Closure-passing optimizes the case of closed functions: they is no need to create a closure—the code pointer can be passed directly Steckler and Wand (1997). However, its impact on the type translations is more important: the closure-passing variant requires more type-theoretic machinery (*recursive types* and *rows*).

The closure-passing variant is as follows:

$$\begin{array}{rcl}
[\![\lambda x.\, a]\!] & = & \mathsf{let}\ \mathit{code} = \lambda(\mathit{clo}, x).\, \mathsf{let}\ (_, x_1, \ldots, x_n) = \mathit{clo}\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\
 & & (\mathit{code}, x_1, \ldots, x_n) \\[1ex]
[\![a_1\ a_2]\!] & = & \mathsf{let}\ \mathit{clo} = [\![a_1]\!]\ \mathsf{in} \\
 & & \mathsf{let}\ \mathit{code} = \mathsf{proj}_0\ \mathit{clo}\ \mathsf{in} \\
 & & \mathit{code}\ (\mathit{clo}, [\![a_2]\!])
\end{array}$$

where $\{x_1, \ldots, x_n\}$ is $\mathsf{fv}(\lambda x.\, a)$ (the variables *code* and *clo* must be suitably fresh). Note that the layout of the environment must be known only at the closure allocation site, not at the call site. In particular, $\mathsf{proj}_0\ clo$ need not know the size of *clo*.

The environment-passing variant is as follows:

$$\begin{array}{rcl} [\![\lambda x.\, a]\!] & = & \mathsf{let}\ code = \lambda(env, x).\, \mathsf{let}\ (x_1, \ldots, x_n) = env\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\ & & (code, (x_1, \ldots, x_n)) \\ \\ [\![a_1\ a_2]\!] & = & \mathsf{let}\ (code, env) = [\![a_1]\!]\ \mathsf{in} \\ & & code\ (env, [\![a_2]\!]) \end{array}$$

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\lambda x.\, a)$.

To understand type-preserving closure conversion, let us first focus on the environment-passing variant. How can closure conversion be made *type-preserving*? The key issue is to find a sensible definition of the type translation. In particular, what is the translation of a function type, $[\![\tau_1 \to \tau_2]\!]$? Let us examine the closure allocation code again. Suppose $\Gamma \vdash \lambda x.\, a : \tau_1 \to \tau_2$. Suppose, without loss of generality (see Remark 5), that $\mathsf{dom}(\Gamma)$ is exactly $\mathsf{fv}(\lambda x.\, a)$, *i.e.* $\{x_1, \ldots, x_n\}$. If Γ is $x_1 : \tau'_1; \ldots; x_n : \tau'_n$, we write $[\![\Gamma]\!]$ for $x_1 : [\![\tau'_1]\!]; \ldots; x_n : [\![\tau'_n]\!]$. By abuse of notation, we also use $[\![\Gamma]\!]$ in a type position to mean the tuple type $[\![\tau'_1]\!] \times \ldots \times [\![\tau'_n]\!]$.

By hypothesis, we have $[\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![a]\!] : [\![\tau_2]\!]$, so *env* has type $[\![\Gamma]\!]$, *code* has type $([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$, and the entire closure has type $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$. So, can we adopt $(([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times [\![\Gamma]\!]$ as a definition of $[\![\tau_1 \to \tau_2]\!]$?

Naturally not. This definition is mathematically ill-formed, as we cannot use Γ out of the blue! That is, we cannot have a translation of $[\![\tau_1 \to \tau_2]\!]$ that depends on the type of free variables of a! Indeed. *we need a uniform translation of types,* not just because it is nice to have one, but because it describes a *uniform calling convention.* If closures with distinct environment sizes or layouts receive distinct types, then we will be unable to translate well-typed code: $\mathsf{if}\ \ldots\ \mathsf{then}\ \lambda x.\, x + y\ \mathsf{else}\ \lambda x.\, x$. Furthermore, we want function invocations to be translated uniformly, without knowledge of the size and layout of the closure's environment.

The only sensible solution is: $\exists \alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha$. An *existential quantification* over the type of the environment abstracts away the differences in size and layout. Enough information is retained to ensure that the application of the code to the environment is valid: this is expressed by letting the variable α occur twice on the right-hand side.

The existential quantification also provides a form of *security.* The caller cannot do anything with the environment except pass it as an argument to the code. In particular, it cannot inspect or modify the environment. For instance, in the source language, the following coding style guarantees that x remains even, no matter how f is used:

$$\mathsf{let}\ f = \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda().\, x := (!\, x + 2); !\, x$$

After closure conversion, the reference x is reachable via the closure of f. A malicious, untyped client could write an odd value to x. However, a *well-typed* client is unable to do so. This encoding is not just type-preserving, but also *fully abstract:* it preserves (a typed

version of) observational equivalence (Ahmed and Blume, 2008).

Remark 5 In order to support the hypothesis $\mathsf{dom}(\Gamma) = \mathsf{fv}(\lambda x.\, a)$ at every λ-abstraction, it is possible to introduce an (admissible) *weakening* rule:

$$\frac{\Gamma_1; \Gamma_2 \vdash a : \tau \qquad x \# a}{\Gamma_1; x : \tau'; \Gamma_2 \vdash a : \tau}\ \textsc{Weakening}$$

If the weakening rule is applied eagerly at every λ-abstraction, then the hypothesis is met, and closures have *minimal environments.* (In some cases, one may not use minimal environments, *e.g.* to allow sharing of environments between several closures.)

6.2 Existential types

One can extend System F with *existential types,* in addition to universals:

$$\tau ::= \ldots \mid \exists\alpha.\tau$$

As in the case of universals, there are *type-passing* and *type-erasing* interpretations of the terms and typing rules and, in the latter interpretation, there are *explicit* and *implicit* versions. Let us first look at the type-erasing interpretation with an explicit notation for introducing and eliminating existential types.

6.2.1 Existential types in Church style (explicitly typed)

The existential quantifier are introduced and eliminated as follows:

$$\frac{\Gamma \vdash M : [\alpha \mapsto \tau']\tau}{\Gamma \vdash \mathsf{pack}\ \tau', M\ \mathsf{as}\ \exists\alpha.\,\tau : \exists\alpha.\,\tau}\ \textsc{Pack} \qquad \frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2 : \tau_2}\ \textsc{Unpack}$$

The side condition $\alpha \# \tau_2$ is *mandatory* here to ensure well-formedness of the conclusion. If well-formedness conditions were explicit in judgments, this could be equivalently defined as $\Gamma \vdash \tau_2$, as it would imply $\alpha \# \tau_2$ since the last premise implies $\alpha \# \Gamma$.

Notice the *imperfect* duality between existential and universals, reminded below:

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\,\tau}\ \textsc{TAbs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\,\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}\ \textsc{TApp}$$

This suggests a simpler elimination form, perhaps like this:

$$\frac{\Gamma \vdash M : \exists\alpha.\tau}{\Gamma, \alpha \vdash \mathsf{unpack}\ M : \tau} \qquad \textit{Broken!}$$

Informally, this could mean that, if M has type τ for some *unknown* α, then it has type τ, where α is "fresh". Unfortunately, this is a broken rule, as we could immediately *universally* quantify over α and conclude that $\Gamma \vdash M : \forall\alpha.\tau$. This is nonsense! Replacing the premise $\Gamma, \alpha \vdash M : \exists\alpha.\tau$ by the conjunction $\Gamma \vdash M : \exists\alpha.\tau$ and $\alpha \in \mathsf{dom}(\Gamma)$ would make the rule even more permissive, so it wouldn't help.

A correct elimination rule must force the existential package to be *used* in a way that does not rely on the value of α. Hence, the elimination rule must have control over the *user* or *continuation* of the package—that is, over the term M_2. The restriction $\alpha \# \tau_2$ prevents writing "$\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ x$", which would be equivalent to the unsound "$\mathsf{unpack}\ M$" discussed above. The fact that α is bound within M_2 forces it to be treated abstractly. In fact, M_2 must be *polymorphic* in α. The rule could be written:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash \Lambda\alpha.\lambda x.\, M_2 : \forall\alpha.\, \tau_1 \to \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2 : \tau_2}$$

Or, more economically:

$$\frac{\Gamma \vdash M_1 : \exists\alpha.\tau_1 \qquad \Gamma \vdash M_0 : \forall\alpha.\, \tau_1 \to \tau_2 \qquad \alpha \# \tau_2}{\Gamma \vdash \mathsf{unpack}\ M_1\ M_0 : \tau_2}$$

where M_0 would evaluate to a value of the form $\Lambda\alpha.\lambda x.\, M_2$.

One could even view "unpack" as a *constant* with all the types $(\exists\alpha.\tau_1) \to (\forall\alpha.\, (\tau_1 \to \tau_2)) \to \tau_2$. or, letting β range over τ_2, all types $\forall\beta.\, (\exists\alpha.\tau) \to (\forall\alpha.\, (\tau \to \beta)) \to \beta$ or even better, $\exists\alpha.\tau \to \forall\beta.\, ((\forall\alpha.\, (\tau \to \beta)) \to \beta)$, since β should not occur free in τ. We thus introduce a *family* of constants "$\mathsf{unpack}_{\exists\alpha.\tau}$" with type $\exists\alpha.\tau \to \forall\beta.\, ((\forall\alpha.\, (\tau \to \beta)) \to \beta)$. Notice that the variable β, which stands for τ_2, is bound prior to α, so it naturally cannot be instantiated to a type that refers to α. This reflects the side condition $\alpha \# \tau_2$. If desired, "$\mathsf{pack}_{\exists\alpha.\tau}$" could also be viewed as a constant of type $\forall\alpha.\, (\tau \to \exists\alpha.\tau)$. Similarly, we may introduce a constant pack with all the types $[\alpha \mapsto \tau']\tau \to \exists\alpha.\tau$, which we may factor as the following types $\forall\alpha.\, (\tau \to \exists\alpha.\tau)$.

In summary, System F with existential types can also be presented by introducing two families of constants of constants with the following types:

$$\mathsf{pack}_{\exists\alpha.\tau} : \ \forall\alpha.\, (\tau \to \exists\alpha.\tau) \qquad \mathsf{unpack}_{\exists\alpha.\tau} : \ \exists\alpha.\, \tau \to \forall\beta.\, ((\forall\alpha.\, (\tau \to \beta)) \to \beta) \qquad (\Delta_\exists)$$

These can be read as follows: for *any* α, if you have a τ, then, for *some* α, you have a τ; conversely, if, for *some* α, you have a τ, then, (for any β,) if you wish to obtain a β out of $\exists\alpha.\, \tau$, you must present a function which, for *any* α, obtains a β out of a τ. This is somewhat reminiscent of ordinary first-order logic: $\exists x.F$ is equivalent to, and can be defined as, $\neg(\forall x.\, \neg F)$.

One can go one step further and entirely encode existential types into universal types. This encoding is actually a small example of type-preserving translation! The type transla-

tion is *double negation:*

$$[\![\exists\alpha.\tau]\!] \;=\; \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \qquad \text{if } \beta \# \tau$$

There is actually little choice for the term translation, if the translation is to be type-preserving:

$$\begin{aligned}
[\![\mathsf{pack}_{\exists\alpha.\tau}]\!] \;&:\; \forall\alpha.\,([\![\tau]\!] \to [\![\exists\alpha.\tau]\!]) \\
&=\; \Lambda\alpha.\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x \\
[\![\mathsf{unpack}_{\exists\alpha.\tau}]\!] \;&:\; [\![\exists\alpha.\tau]\!] \to \forall\beta.\,((\forall\alpha.\,([\![\tau]\!] \to \beta)) \to \beta) \\
&=\; \lambda x{:}[\![\exists\alpha.\tau]\!].\,x
\end{aligned}$$

This encoding is a *continuation-passing transform*. This encoding is due to Reynolds 1983, although it has more ancient roots in logic.

When existential are presented as constrants, their semantics is defined by seeing $\mathsf{pack}_{\exists\alpha.\tau}$ as a unary constructor and $\mathsf{unpack}_{\exists\alpha.\tau}$ as a unary destructor with the following reduction rule:

$$\mathsf{unpack}_{\exists\alpha.\tau_0}\,(\mathsf{pack}_{\exists\alpha.\tau}\;\tau'\;V) \;\longrightarrow\; \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\;\tau'\;V \qquad (\delta_\exists)$$

Exercise 42 *Show that this δ-rule satisfies the progress and subject reduction assumptions for constants with the types in $\Delta_\exists$. (You may assume that the standard lemmas still hold.)* *(Solution p. 169)* □

Exercise 43 *The $\delta_\exists$ reduction for existential is permissive it allows reducing of ill-typed terms. Give a more restrictive version of the rule. What will need to be changed in the proof of subject reduction and proress for the δ-rule (Exercise 42)?* *(Solution p. 169)* □

Notice that our $\delta_\exists$-reduction reduces an "unpack of a pack" to a polymorphic function that applies its argument to the packed value. This is still a form of continuation-passing-style encoding. It seems more natural to treat $\mathsf{unpack}_{\exists\alpha.\tau}$ as a binary destructor to avoid this intermediate step and have the more intuitive reduction rule:

$$\mathsf{unpack}_{\exists\alpha.\tau_0}\,(\mathsf{pack}_{\exists\alpha.\tau}\;\tau'\;V)\;\tau_1\;(\Lambda\alpha.\lambda x{:}\tau.\,M) \;\longrightarrow\; [x \mapsto V][\alpha \mapsto \tau']M \qquad (\delta_\exists)$$

However, this does not fit in our framework and notion of arity for constants where all type arguments must be passed first and not interleaved with value arguments. Our framework could be extended to the above δ-rules for existentials, but the presentation would become cumbersome.

Alternatively, if existential are primitive, their semantics is defined by extending values and evaluation contexts as follows:

$$V \;::=\; \ldots \mid \mathsf{pack}\;\tau', V\;\mathsf{as}\;\tau \qquad E \;::=\; \ldots \mid \mathsf{pack}\;\tau', [\,]\;\mathsf{as}\;\tau \;\mid\; \mathsf{let}\;\alpha, x = \mathsf{unpack}\;[\,]\;\mathsf{in}\;M$$

and by adding the following reduction rule:

$$\mathsf{let}\ \alpha, x = \mathsf{unpack}\ (\mathsf{pack}\ \tau', V\ \mathsf{as}\ \tau)\ \mathsf{in}\ M \longrightarrow [\alpha \mapsto \tau'][x \mapsto V]M$$

Exercise 44 *Check that the proofs of subject reduction and progress for System* F *extend to existential types. (Just check the new cases, assuming that the standard lemmas still hold.)* □

The reduction rule for existential destructs its arguments. Hence, $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ cannot be reduced unless M_1 is itself a packed expression, which is indeed the case when M_1 is a value (or in head normal form). This contrasts with $\mathsf{let}\ x : \tau = M_1\ \mathsf{in}\ M_2$ where M_1 need not be evaluated and may be an application (*e.g.* in call-by-name or with strong reduction).

Exercise 45 *The reduction of* $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ *could be problematic when* M_1 *is not a value. Illustrate this on an example* (You may use the following hint if needed: *lanoitidnocaesu.)* *(Solution p. 169)* □

One may wonder whether the pack construct is not too verbose: isn't the type witness type annotation τ' in rule Pack superfluous? The type τ_0 of M is fully determined by M and the given type $\exists\alpha.\tau$ of the packed value. Checking that τ_0 is of the form $[\alpha \mapsto \tau']\tau$ is the matching problem for second-order types, which is simple. However, the reduction rule need the witness type τ'. If it were not available, it would have to be computed during reduction. The reduction rule would then not be pure rewriting. The explicitly-typed language need the witness type for simplicity, while in the surface language, it could be omitted and reconstructed by second-order matching.

6.2.2 Implicitly-typed existential types

Intuitively, pack and unpack are just type information that can be dropped by type erasure. More precisely, the erasure of $\mathsf{pack}\ \tau', M\ \mathsf{as}\ \exists\alpha.\,\tau\exists\alpha.\,\tau$ is M and the erasure of $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$ is a let-binding $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$. After type-erasure, the following typing rules for existential types in implicit-typed System F:

$$\textsc{if-Unpack}\quad \frac{\Gamma \vdash a_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \ \#\ \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2} \qquad\qquad \textsc{if-Pack}\quad \frac{\Gamma \vdash a : [\alpha \mapsto \tau']\tau}{\Gamma \vdash a : \exists\alpha.\tau}$$

Notice, that the let-binding is not typechecked as syntactic sugar for an immediate application. Its semantics remains the same.

$$E ::= \ldots \mathsf{let}\ x = [\,]\ \mathsf{in}\ M \qquad\qquad \mathsf{let}\ x = V\ \mathsf{in}\ M \longrightarrow [x \mapsto V]M$$

Is the semantics still type-erasing? Yes, it is, but there is a subtlety! This is only true in call-by-value. In a call-by-name semantics, a let-bound expression is not reduced prior to

substitution of the argument, that is, the rule would be:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \longrightarrow [x \mapsto a_1]a_2$$

With existential types, this breaks subject reduction! This was first noticed by Sørensen and Urzyczyn (2006). See also (Fujita and Schubert, 2009, §9).

To see this, let τ_0 be $\exists\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$ and let v_0 be a value of type bool. Then, let v_1 and v_2 two values of type τ_0 with incompatible witness types, taking for instance, $\lambda f.\,\lambda x.\,1{+}(f\ (1+x))$ and $\lambda f.\,\lambda x.\,not\ (f\ (not\ x))$. Let v be the function $\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ v_1\ \mathsf{else}\ v_2$ of type $\mathsf{bool} \to \tau_0$, which returns either one of V_1 or V_2 depending on its argument b. We then have the reduction

$$a_1 \;=\; \mathsf{let}\ x = v\ v_0\ \mathsf{in}\ x\ (x\ (\lambda y.\,y)) \;\longrightarrow\; v\ v_0\ (v\ v_0\ (\lambda y.\,y)) \;=\; a_2$$

The typing judgment $\varnothing \vdash a_1 : \exists\alpha.\,\alpha \to \alpha$ holds, while $\varnothing \vdash a_2 : \tau$ does not hold for any τ. Indeed, the term a_1 is well-typed since $v\ v_0$ has type τ_0, hence x can be assumed of type $(\beta \to \beta) \to (\beta \to \beta)$ for some unknown type β and $\lambda y.\,y$ is of type $\beta \to \beta$. However, without the outer existential type $v\ v_0$ can only be typed with $(\forall\alpha.\,\alpha \to \alpha) \to \exists\alpha.\,(\alpha \to \alpha)$, because the value returned by the function need different witnesses for α. This is demanding too much on its argument and the outer application is ill-typed.

One may wonder whether the syntax should not allow the *implicit* introduction of unpacking instead. For instance, one could argue that if some expression is the expansion of a well-typed let-binding, then it should also be well-typed:

$$\frac{\Gamma \vdash a_1 : \exists\alpha.\tau_1 \qquad \Gamma, \alpha, x : \tau_1 \vdash a_2 : \tau_2 \qquad \alpha \mathrel{\#} \tau_2}{\Gamma \vdash [x \mapsto a_1]a_2 : \tau_2}$$

However, this rule is not quite satisfactory as it does not have a logical flavor. Moreover, it fixes the previous example, but does not help with the general case: Pick a_1 that is not yet a value after one reduction step. Then, after let-expansion reduce one of the two occurrences of a_1. The result is no longer of the form $[x \mapsto a_1]a_2$.

In summary, existential types are tricky: The subject reduction property breaks if reduction is not restricted to expressions in head-normal forms. Unrestricted reduction is still safe because well-typedness may eventually be recovered by further reduction steps—so that progress will never break.

Interestingly, the CPS encoding of existential types (1) enforces the evaluation of the packed value (2) before it can be unpacked (3) and substituted(4):

$$\begin{array}{rcll}
[\![\mathsf{unpack}\ a_1\ (\lambda x.\,a_2)]\!] & = & [\![a_1]\!]\ (\lambda x.\,[\![a_2]\!]) & \textbf{(1)}\\
& \longrightarrow & (\lambda k.\,[\![a]\!]\ k)\ (\lambda x.\,[\![a_2]\!]) & \textbf{(2)}\\
& \longrightarrow & (\lambda x.\,[\![a_2]\!])\ [\![a]\!] & \textbf{(3)}\\
& \longrightarrow & [x \mapsto [\![a]\!]][\![a_2]\!] & \textbf{(4)}
\end{array}$$

In the call-by-value setting, $\lambda k.\,[\![a]\!]\ k$ would come from the reduction of $[\![\mathsf{pack}\ a]\!]$, *i.e.* is $(\lambda k.\,\lambda x.\,k\ x)\ [\![a]\!]$, so that a is always a value v. However, a need not be a value. What is

essential is again that a_1 be reduced to some head normal form $\lambda k. [\![a]\!]\ k$.

6.2.3 Existential types in ML

What if one wished to extend ML with existential types? Full type inference for existential types is undecidable, just like type inference for universals. However, introducing existential types in ML is easy if one is willing to rely on user-supplied *annotations* that indicate where to pack and unpack.

This *iso-existential* approach was suggested by Läufer and Odersky (1994). Iso-existential types are explicitly *declared*, much as datatypes:

$$D\ \vec{\alpha} \approx \exists\bar{\beta}.\tau \qquad \text{if } \mathsf{ftv}(\tau) \subseteq \bar{\alpha} \cup \bar{\beta} \quad \text{and} \quad \bar{\alpha} \# \bar{\beta}$$

This introduces two constants, with the following type schemes:

$$\mathsf{pack}_D : \forall\bar{\alpha}\bar{\beta}.\tau \to D\ \vec{\alpha} \qquad\qquad \mathsf{unpack}_D : \forall\bar{\alpha}\gamma. D\ \vec{\alpha} \to (\forall\bar{\beta}.(\tau \to \gamma)) \to \gamma$$

(Compare with basic iso-recursive types, where $\bar{\beta} = \varnothing$.)

Unfortunately, the "type scheme" of unpack_D is *not* an ML type scheme. A solution is to make unpack_D a binary primitive construct, rather than a constant, with an *ad hoc* typing rule:

$$\textsc{Unpack}_D\quad \frac{\Gamma \vdash M_1 : D\ \vec{\tau} \qquad \Gamma \vdash M_2 : \forall\bar{\beta}.([\vec{\alpha} \mapsto \vec{\tau}]\tau \to \tau_2) \qquad \bar{\beta} \# \vec{\tau}, \tau_2}{\Gamma \vdash \mathsf{unpack}_D\ M_1\ M_2 : \tau_2} \qquad \text{where } D\ \vec{\alpha} \approx \exists\bar{\beta}.\tau$$

We have seen a version of this rule in System F earlier; this in an ML version. The term M_2 must be polymorphic, which Gen can prove.

Iso-existential types are perfectly compatible with ML type inference. The constant pack_D admits an ML type scheme, so it is not problematic. The construct unpack_D leads to this constraint generation rule (cf. §5):

$$\langle\!\langle \mathsf{unpack}_D\ M_1\ M_2 : \tau_2 \rangle\!\rangle \;=\; \exists\bar{\alpha}.\big(\ \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \wedge \forall\bar{\beta}.\langle\!\langle M_2 : \tau \to \tau_2 \rangle\!\rangle\ \big)$$

where $D\ \vec{\alpha} \approx \exists\bar{\beta}.\tau$ and, *w.l.o.g.*, $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$. Note that a universally quantified constraint appears where polymorphism is *required.*

In practice, Läufer and Odersky suggest fusing iso-existential types with algebraic data types. The somewhat bizarre Haskell syntax for this is:

$$\mathsf{data}\ D\ \vec{\alpha} = \mathsf{forall}\ \bar{\beta}.\ell\ \tau$$

where ℓ is a data constructor. The elimination construct $\langle\!\langle \mathsf{case}\ M_1\ \mathsf{of}\ \ell\ x \to M_2 : \tau_2 \rangle\!\rangle$ and is typed as follows:

$$\langle\!\langle \mathsf{case}\ M_1\ \mathsf{of}\ \ell\ x \to M_2 : \tau_2 \rangle\!\rangle \;=\; \exists\bar{\alpha}.\big(\ \langle\!\langle M_1 : D\ \vec{\alpha} \rangle\!\rangle \wedge \forall\bar{\beta}.\mathsf{def}\ x : \tau\ \mathsf{in}\ \langle\!\langle M_2 : \tau_2 \rangle\!\rangle\ \big)$$

where, w.l.o.g., $\bar{\alpha}\bar{\beta} \# M_1, M_2, \tau_2$.

Examples Define $\mathsf{Any} \approx \exists\beta.\beta$. The following code that attempts to extract the raw content of a package fails:

$$\langle\!\langle \mathsf{unpack}_{\mathsf{Any}}\ M_1\ (\lambda x.\,x) : \tau_2 \rangle\!\rangle \ = \ \langle\!\langle M_1 : \mathsf{Any} \rangle\!\rangle \wedge \forall\beta.\,\langle\!\langle \lambda x.\,x : \beta \to \tau_2 \rangle\!\rangle \ \Vdash \ \forall\beta.\,\beta = \tau_2 \ \equiv \ \mathsf{false}$$

Now, define $D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$. A client that regards β as abstract succeeds:

$$\begin{array}{rl} & \langle\!\langle \mathsf{unpack}_D\ M_1\ (\lambda(f,y).\,f\ y) : \tau \rangle\!\rangle \\ = & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\langle\!\langle \lambda(f,y).\,f\ y : ((\beta \to \alpha) \times \beta) \to \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\mathsf{def}\ f : \beta \to \alpha; y : \beta\ \mathsf{in}\ \langle\!\langle f\ y : \tau \rangle\!\rangle) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \forall\beta.\,\tau = \alpha) \\ \equiv & \exists\alpha.(\langle\!\langle M_1 : D\ \alpha \rangle\!\rangle \wedge \tau = \alpha) \\ \equiv & \langle\!\langle M_1 : D\ \tau \rangle\!\rangle \end{array}$$

Remark 6 We reuse the type $D\ \alpha \approx \exists\beta.(\beta \to \alpha) \times \beta$ of frozen computations, defined above. Assume given a list l of elements of type $D\ \tau_1$. Assume given a function g of type $\tau_1 \to \tau_2$. We may transform the list into a new list l' of frozen computations of type $D\ \tau_2$ (without actually running any computation).

```
List.map (λ(z) let D(f, y) = z in D((λ(z) g (f z)), y))
```

We may generalize the code into a functional that receives g and and l as arguments and returns l'. Unfortunately, the following code does not typecheck:

```
let lift g l = List.map (λ(z) let D(f, y) = z in D((λ(z) g (f z)), y))
```

The problem is that, in expression $\mathsf{let}\ \alpha, x = \mathsf{unpack}\ M_1\ \mathsf{in}\ M_2$, occurrences of x can only be passed to polymorphic functions so that the type α of x does not escape from its scope. That is first-class existential types calls for first-class universal types as well!

Mitchell and Plotkin (1988) note that existential types offer a means of explaining *abstract types.* For instance, the type:

$$\exists\mathsf{stack}.\{\ \mathit{empty} : \mathsf{stack};\ \ \mathit{push} : \mathsf{int} \times \mathsf{stack} \to \mathsf{stack};\ \ \mathit{pop} : \mathsf{stack} \to \mathsf{option}\ (\mathsf{int} \times \mathsf{stack})\ \}$$

specifies an abstract implementation of integer stacks.

Unfortunately, it was soon noticed that the elimination rule is too awkward, and that existential types alone do not allow designing *module systems* Harper and Pierce (2005). Montagu and Rémy (2009) make existential types *more flexible* in several important ways, and argue that they might explain modules after all.

6.2.4 Existential types in OCaml

Amusingly, existential types were first available in OCaml via abstract types and first-class modules. There are now also available as a degenerate case of Generalized Algebraic DataTypes (GADT) which coincides with the appraoached described above.

For example, one may defined the previous datatype of frozen computations:

```
type 'a d = D : ('b → 'a) * 'b → 'a d
let freeze f x = D (f, x)
let run (D (f, x)) = f x
```

Here is the equivalent, more verbose code with modules:

```
module type D = sig type b type a val f : b → a val x : b end
let freeze (type u) (type v) f  x =
   (module struct type b =  u type a = v  let f = f let x = x end : D);;
let unfreeze (type u) (module M : D with type a = u) = M.f M.x
```

6.3 Typed closure conversion

Equipped with existential types, we may now revisit type closure conversion.

6.3.1 Environment-passing closure conversion

Remember that we came to the conclusion that the translation of arrow types $[\![\tau_1 \to \tau_2]\!]$ must be $\exists\alpha.((\alpha\times[\![\tau_1]\!]) \to [\![\tau_2]\!])\times\alpha$. Let us show that we may translate expressions so as to preserve well-typedness, *i.e.* so that $\Gamma \vdash M : \tau$ implies $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$. Assume $\Gamma \vdash \lambda x.\, M : \tau_1 \to \tau_2$ and $\mathsf{dom}(\Gamma) = \{x_1, \dots x_n\} = \mathsf{fv}(\lambda x{:}\tau_1.\, M)$. We may now hide the dependence on Γ using an existential type:

$$\begin{array}{rcl} [\![\lambda x{:}\tau_1.\, M]\!] & = & \mathsf{let}\ \mathit{code} : ([\![\Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\ & & \quad \lambda(\mathit{env} : [\![\Gamma]\!],\, x : [\![\tau_1]\!]).\, \mathsf{let}\ (x_1, \dots x_n : [\![\Gamma]\!]) = \mathit{env}\ \mathsf{in}\ [\![M]\!]\ \mathsf{in} \\ & & \mathsf{pack}\ [\![\Gamma]\!], (\mathit{code}, (x_1, \dots, x_n))\ \mathsf{as}\ \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha \\ & : & \exists\alpha.((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha \quad = \quad [\![\tau_1 \to \tau_2]\!] \end{array}$$

In the case of application, assume $\Gamma \vdash M : \tau_1 \to \tau_2$ and $\Gamma \vdash M_1 : \tau_1$ and take:

$$\begin{array}{rcl} [\![M\ M_1]\!] & = & \mathsf{let}\ \alpha, (\mathit{code} : (\alpha \times [\![\tau_1]\!]) \to \tau_2,\ \mathit{env} : \alpha) = \mathsf{unpack}\ [\![M]\!]\ \mathsf{in}\ \mathit{code}\ (\mathit{env},\ [\![M_1]\!]) \\ & : & [\![\tau_2]\!] \end{array}$$

For *recursive functions* we may use the "fix-code" variant (Morrisett and Harper, 1998):

$$\begin{array}{rcl} [\![\mu f.\lambda x.a]\!] & = & \mathsf{let\ rec}\ \mathit{code}\ (\mathit{env}, x) = \\ & & \quad \mathsf{let}\ f = \mathsf{pack}\ (\mathit{code},\ \mathit{env})\ \mathsf{in}\ \mathsf{let}\ (x_1, \dots, x_n) = \mathit{env}\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\ & & \mathsf{pack}\ (\mathit{code}, (x_1, \dots, x_n)) \end{array}$$

where $\{x_1, \dots, x_n\} = \mathsf{fv}(\mu f.\lambda x.a)$. The translation of applications is unchanged as recursive and non-recursive functions have an identical calling convention. This translation builds recursive code, avoiding a recursive closure, hence the code is easy to type. Unfortunately, as a counterpart, a new closure is allocated at every call, which is the weak point of this variant.

Instead, the "fix-pack" variant (Morrisett and Harper, 1998) uses an extra field in the environment to store a back pointer to the closure:

$$\begin{aligned}[\![\mu f.\lambda x.a]\!] \quad = \quad & \mathsf{let}\ code = \lambda(env, x).\, \mathsf{let}\ (f, x_1, \dots, x_n) = env\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\ & \mathsf{let\ rec}\ \ clo = (code, (clo, x_1, \dots, x_n))\ \mathsf{in}\ clo\end{aligned}$$

where $\{x_1, \dots, x_n\} = \mathsf{fv}(\mu f.\lambda x.a)$. Hence, we avoid rebuilding the closure at every call by creating a recursive closure. However, this requires, in general, recursively-defined *values* and closures are now *cyclic* data structures.

Here is how the "fix-pack" variant is type-checked. Assume $\Gamma \vdash \mu f : \tau_1 \to \tau_2.\lambda x.M : \tau_1 \to \tau_2$ and $\mathsf{dom}(\Gamma) = \{x_1, \dots, x_n\} = \mathsf{fv}(\mu f.\lambda x.M)$.

$$\begin{aligned}& [\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \\ & \quad \mathsf{let}\ code : ([\![f : \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\ & \qquad \lambda(env : [\![f : \tau_1 \to \tau_2, \Gamma]\!],\ x : [\![\tau_1]\!]).\, \mathsf{let}\ (f, x_1, \dots, x_n) : [\![f : \tau_1 \to \tau_2, \Gamma]\!] = env\ \mathsf{in}\ [\![M]\!]\ \mathsf{in} \\ & \quad \mathsf{let\ rec}\ clo : [\![\tau_1 \to \tau_2]\!] = \\ & \qquad \mathsf{pack}\ [\![f : \tau_1 \to \tau_2, \Gamma]\!], (code, (clo, x_1, \dots, x_n))\ \mathsf{as}\ \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\ & \quad \mathsf{in}\ clo\end{aligned}$$

This implements monomorphic recursion, as by default in ML. To allow the recursive function to be polymorphic, we can generalize the encoding afterwards:

$$[\![\Lambda\vec{\beta}.\mu f : \tau_1 \to \tau_2.\lambda x.M]\!] = \Lambda\vec{\beta}.[\![\mu f : \tau_1 \to \tau_2.\lambda x.M]\!]$$

whenever the right-hand side is well-defined. This allows the *indirect* compilation of polymorphic recursive functions as long as the recursion is monomorphic.

Fortunately, the encoding can be straightforwardly adapted to *directly* compile polymorphically recursive functions into polymorphic closure.

$$\begin{aligned}& [\![\mu f : \forall\vec{\beta}.\, \tau_1 \to \tau_2.\, \lambda x.M]\!] = \\ & \quad \mathsf{let}\ code : \forall\vec{\beta}.\, ([\![f : \forall\vec{\beta}.\, \tau_1 \to \tau_2; \Gamma]\!] \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\ & \qquad \Lambda\bar{\beta}.\lambda(env : [\![f : \forall\vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!],\ x : [\![\tau_1]\!]). \\ & \qquad\quad \mathsf{let}\ (f, x_1, \dots, x_n) : [\![f : \forall\vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!] = env\ \mathsf{in}\ [\![M]\!]\ \mathsf{in} \\ & \quad \mathsf{let\ rec}\ clo : [\![\forall\vec{\beta}.\, \tau_1 \to \tau_2]\!] = \Lambda\vec{\beta}. \\ & \qquad \mathsf{pack}\ [\![f : \forall\vec{\beta}.\, \tau_1 \to \tau_2, \Gamma]\!], (code\ \vec{\beta}, (clo, x_1, \dots, x_n))\ \mathsf{as}\ \exists\alpha((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]) \times \alpha) \\ & \quad \mathsf{in}\ clo\end{aligned}$$

In summary, the environment-passing closure conversion is simple, but it requires the introduction of recursive non-functional values $\mathsf{let\ rec}\ \ x = V\ \mathsf{in}\ M$. While this is a useful construct, it really alters the operational semantics and requires updating the type soundness proof (as recursive non-functional values were not permitted so far).

6.3.2 Closure-passing closure conversion

Recall the *closure-passing* variant:

$$\begin{array}{rcl}
[\![\lambda x.\, a]\!] & = & \mathsf{let}\ \mathit{code} = \lambda(\mathit{clo}, x).\, \mathsf{let}\ (_, x_1, \ldots, x_n) = \mathit{clo}\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\
 & & (\mathit{code}, x_1, \ldots, x_n) \\
[\![a_1\ a_2]\!] & = & \mathsf{let}\ \mathit{clo} = [\![a_1]\!]\ \mathsf{in}\ \mathsf{let}\ \mathit{code} = \mathsf{proj}_0\ \mathit{clo}\ \mathsf{in}\ \mathit{code}\ (\mathit{clo}, [\![a_2]\!])
\end{array}$$

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\lambda x.\, a)$.

There are two difficulties to typecheck this: first, a closure is a tuple, whose *first* field—the code pointer—should be *exposed*, while the number and types of the remaining fields—the environment—should be abstract; second, the first field of the closure contains a function that expects *the closure itself* as its first argument.

To describe this, we use two type-theoretic mechanisms; first existential quantification over the *tail* of a tuple (a.k.a. a *row*) to allow the environment to remain abstract; and *recursive types* to allow the closure to points to itself.

Tuples, rows, row variables Let us first introduce extensible tuples. The standard tuple types that we have used so far are:

$$\begin{array}{rcll}
\tau & ::= & \ldots \mid \Pi\, R & \text{– types} \\
R & ::= & \epsilon \mid (\tau; R) & \text{– rows}
\end{array}$$

The notation $(\tau_1 \times \ldots \times \tau_n)$ was sugar for $\Pi\ (\tau_1; \ldots; \tau_n; \epsilon)$. Let us introduce *row variables* and allow *quantification* over them:

$$\begin{array}{rcll}
\tau & ::= & \ldots \mid \Pi\, R \mid \forall \rho.\, \tau \mid \exists \rho. \tau & \text{– types} \\
R & ::= & \rho \mid \epsilon \mid (\tau; R) & \text{– rows}
\end{array}$$

This allows reasoning about the first few fields of a tuple whose length is not known. The typing rules for tuple construction and deconstruction are:

$$\textsc{Tuple}\ \frac{\forall i.\ \in [1, n] \quad \Gamma \vdash M_i : \tau_i}{\Gamma \vdash (M_1, \ldots, M_n) : \Pi\ (\tau_1; \ldots; \tau_n; \epsilon)} \qquad \textsc{Proj}\ \frac{\Gamma \vdash M : \Pi\ (\tau_1; \ldots; \tau_i; R)}{\Gamma \vdash \mathsf{proj}_i\ M : \tau_i}$$

These rules make sense with or without row variables. Projection does not care about the fields beyond i. Thanks to row variables, this can be expressed in terms of *parametric polymorphism:* $\mathsf{proj}_i : \forall \alpha_1 \ldots \alpha_i \rho.\, \Pi\ (\alpha_1; \ldots \alpha_i; \rho) \to \alpha_i$.

Remark 7 Rows were invented by Wand (1988) and improved by Rémy (1994b) in order to ascribe precise types to operations on *records.* The case of tuples, presented here, is simpler. Rows are used to describe *objects* in OCaml (Rémy and Vouillon, 1998). Rows are explained in depth by Pottier and Rémy (2005).

Back to closure-passing closure conversion Rows and recursive types allow to define the translation of types in the closure-passing variant:

$$[\![\tau_1 \to \tau_2]\!] \quad = \quad \exists \rho.\mu\alpha.\Pi\ (((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!])\,;\rho)$$

ρ describes the environment represented as a row of fields, which is abstract; α is the concrete type of the closure that is to refer to recursively; $\Pi\ (((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!])\,;\rho)$ is a tuple that begins with a code pointer of type $(\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]$ and continues with the environment ρ. See the “fix-type” encoding proposed by Morrisett and Harper (1998).

Notice that the type is $\exists\rho.\ \mu\alpha.\ \tau$ and not $\mu\alpha.\ \exists\rho.\ \tau$: The type of the environment is fixed once for all and does not change at each recursive call. Notice that ρ appears only once, which may seem surprising. Usually, an existential type variable appears both at positive and negative occurrences. Here, the variable α appear only at a negative occurrence, but in a recursive part of the type that can be unfolded.

To help checking well-typedness of the encoding, let $\mathsf{Clo}(R)$ abbreviate the concrete type of a closure of row R and $\mathsf{UClo}(R)$ its unfolded version:

$$\begin{array}{rcl} \mathsf{Clo}(R) & \triangleq & \mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R) \\ \mathsf{UClo}(R) & \triangleq & \Pi\ ((\mathsf{Clo}(R) \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R) \end{array}$$

The encoding of arrow types $[\![\tau_1 \to \tau_2]\!]$ is $\exists\rho.\mathsf{Clo}(\rho)$. The encoding of abstactions and applications is:

$$\begin{array}{rcl} [\![\lambda x\!:\!\tau_1.\,M]\!] & = & \mathsf{let}\ \mathit{code} : (\mathsf{Clo}([\![\Gamma]\!]) \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \\ & & \quad \lambda(\mathit{clo} : \mathsf{Clo}([\![\Gamma]\!],\ x : [\![\tau_1]\!]). \\ & & \qquad \mathsf{let}\ (\_, x_1, \ldots, x_n) : \mathsf{UClo}[\![\Gamma]\!] = \mathsf{unfold}\ \mathit{clo}\ \mathsf{in}\ [\![M]\!]\ \mathsf{in} \\ & & \mathsf{pack}\ [\![\Gamma]\!], (\mathsf{fold}\ (\mathit{code}, x_1, \ldots, x_n))\ \mathsf{as}\ \exists\rho.\,\mathsf{Clo}(\rho) \end{array}$$

$$\begin{array}{rcl} [\![M_1\ M_2]\!] & = & \mathsf{let}\ \rho, \mathit{clo} = \mathsf{unpack}\ [\![M_1]\!]\ \mathsf{in} \\ & & \mathsf{let}\ \mathit{code} : (\mathsf{Clo}(\rho) \times [\![\tau_1]\!]) \to [\![\tau_2]\!] = \mathsf{proj}_0\ (\mathsf{unfold}\ \mathit{clo})\ \mathsf{in} \\ & & \mathit{code}\ (\mathit{clo},\ [\![M_2]\!]) \end{array}$$

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\lambda x\!:\!\tau_1.\,M)$.

In the closure-passing variant, recursive functions can be translated as follows:

$$\begin{array}{rcl} [\![\mu f.\lambda x.a]\!] & = & \mathsf{let}\ \mathit{code} = \lambda(\mathit{clo}, x). \\ & & \quad \mathsf{let}\ f = \mathit{clo}\ \mathsf{in}\ \mathsf{let}\ (\_, x_1, \ldots, x_n) = \mathit{clo}\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\ & & (\mathit{code}, x_1, \ldots, x_n) \end{array}$$

where $\{x_1, \ldots, x_n\} = \mathsf{fv}(\mu f.\lambda x.a)$. No extra field or extra work is required to store or construct a representation of the free variable f: the closure itself plays this role. However, this untyped code can only be typechecked when recursion is monomorphic.

Exercise 46 *Carefully check well-typedness of the above translation with monomorphic recursion.* □

To adapt this encoding to polymorphic recursion, the problem is that recursive occurrences of f are rebuilt from the current invocation of the closure, this with the same type since the closure is invoked after type specialization.

By contrast, in the environment passing encoding, the environment contained a polymorphic binding for the recursive calls that was filled with the closure before its invocation, *i.e.* with a polymorphic type.

Fortunately, we may slightly change the encoding, using a recursive closure as in the type-passing version, to allow typechecking in System F.

Remark 8 One could think of changing the encoding of closure types $[\![\tau_1 \to \tau_2]\!]$ to make the encoding work. However, although this should be possible in some more expressive type systems, there seems to be no easy way to do so and certainly not within System F.

Let τ be $\forall\vec{\alpha}.\,\tau_1 \to \tau_2$ and Γ_f be $f:\tau,\Gamma$ where $\vec{\beta} \# \Gamma$

$$
\begin{array}{rcl}
[\![\mu f\!:\!\tau.\,\lambda x.M]\!] & = & \mathsf{let}\ \mathit{code} = \\
& & \quad \Lambda\vec{\beta}.\lambda(\mathit{clo} : \mathsf{Clo}[\![\Gamma_f]\!],\, x : [\![\tau_1]\!]). \\
& & \qquad \mathsf{let}\ (\_\mathit{code}, f, x_1, \ldots, x_n) : \forall\vec{\beta}.\,\mathsf{UClo}([\![\Gamma_f]\!]) = \mathsf{unfold}\ \mathit{clo}\ \mathsf{in}\ [\![M]\!]\ \mathsf{in} \\
& & \mathsf{let\ rec}\ \mathit{clo} : \forall\vec{\beta}.\,\exists\rho.\,\mathsf{Clo}(\rho) = \\
& & \quad \Lambda\vec{\beta}.\mathsf{pack}\ [\![\Gamma]\!], (\mathsf{fold}\ (\mathit{code}\ \vec{\beta}, \mathit{clo}, x_1, \ldots, x_n))\ \mathsf{as}\ \exists\rho.\,\mathsf{Clo}(\rho) \\
& & \ \mathsf{in}\ \mathit{clo}
\end{array}
$$

Remind that $\mathsf{Clo}(R)$ abbreviates $\mu\alpha.\Pi\ ((\alpha \times [\![\tau_1]\!]) \to [\![\tau_2]\!]; R)$. Hence, $\vec{\beta}$ are free variables of $\mathsf{Clo}(R)$. Here, a polymorphic recursive function is *directly* compiled into a polymorphic recursive closure. Notice that the type of closures is unchanged, so the encoding of applications is also unchanged.

Optimizing representations Closure-passing and environment-passing closure conversions cannot be mixed because the calling-convention (*i.e.*, the encoding of application) must be uniform. However, their is some flexibility in the representation of the closure. For instance, the following change is completely local:

$$
\begin{array}{rcl}
[\![\lambda x.\,a]\!] & = & \mathsf{let}\ \mathit{code} = \lambda(\mathit{clo}, x).\,\mathsf{let}\ (\_, (x_1, \ldots, x_n)) = \mathit{clo}\ \mathsf{in}\ [\![a]\!]\ \mathsf{in} \\
& & (\mathit{code}, (x_1, \ldots, x_n))
\end{array}
$$

This allows for sharing the closure (or part of it) may be shared when many definitions share the same closure,

6.3.3 Mutually recursive functions

Can we compile mutually recursive functions $\mu(f_1, f_2).(\lambda x_1.\,a_1, \lambda x_2.\,a_2)$, say a?

The environment passing encoding is as follows:

$$\begin{array}{rcl} [\![a]\!] & = & \text{let } \mathit{code}_i = \lambda(\mathit{env}, x).\, \text{let } (f_1, f_2, x_1, \ldots, x_n) = \mathit{env} \text{ in } [\![a_i]\!] \text{ in} \\ & & \text{let rec } \mathit{env} = (\mathit{clo}_1, \mathit{clo}_2, x_1, \ldots, x_n) \\ & & \quad \text{and } \mathit{clo}_1 = (\mathit{code}_1, \mathit{env}) \\ & & \quad \text{and } \mathit{clo}_2 = (\mathit{code}_2, \mathit{env}) \text{ in} \\ & & \mathit{clo}_1, \mathit{clo}_2 \end{array}$$

Notice that we can share the environment inside the two closures. The closure passing encoding is:

$$\begin{array}{rcl} [\![a]\!] & = & \text{let } \mathit{code}_i = \lambda(\mathit{clo}, x).\, \text{let } (\_, f_1, f_2, x_1, \ldots, x_n) = \mathit{clo} \text{ in } [\![a_i]\!] \text{ in} \\ & & \text{let rec } \mathit{clo}_1 = (\mathit{code}_1, \mathit{clo}_1, \mathit{clo}_2, x_1, \ldots, x_n) \\ & & \quad \text{and } \mathit{clo}_2 = (\mathit{code}_2, \mathit{clo}_1, \mathit{clo}_2, x_1, \ldots, x_n) \\ & & \text{in } \mathit{clo}_1, \mathit{clo}_2 \end{array}$$

Question: Can we share the closures c_1 and c_2 in case n is large?

Here the environment cannot be shared between the two closures, since they belong to tuples of different size. Unless the runtime, in particular the garbage collector, supports such an operation as returning the tail of a tuple without allocating a new tuple. Then we could write:

$$\begin{array}{rcl} [\![a]\!] & = & \text{let } \mathit{code}_1 = \lambda(\mathit{clo}, x).\, \text{let } (\_, \_, f_1, f_2, x_1, \ldots, x_n) = \mathit{clo} \text{ in } [\![a_1]\!] \text{ in} \\ & & \text{let } \mathit{code}_2 = \lambda(\mathit{clo}, x).\, \text{let } (\_, f_1, f_2, x_1, \ldots, x_n) = \mathit{clo} \text{ in } [\![a_2]\!] \text{ in} \\ & & \text{let rec } \mathit{clo}_1 = (\mathit{code}_1, \mathit{code}_2, \mathit{clo}_1, \mathit{clo}_2, x_1, \ldots, x_n) \\ & & \quad \text{and } \mathit{clo}_2 = \mathit{clo}_1.\mathit{tail} \\ & & \text{in } \mathit{clo}_1, \mathit{clo}_2 \end{array}$$

Here $\mathit{clo}_1.\mathit{tail}$ returns a pointer to the tail $(\mathit{code}_2, \mathit{clo}_1, \mathit{clo}_2, x_1, \ldots, x_n)$ of clo_1 without allocating a new tuple.

Encoding of objects The closure-passing representation of mutually recursive functions is similar to the representation of objects in the object-as-record-of-functions paradigm:

A class definition is an object generator:

$$\text{class } c\ (x_1, \ldots x_q)\ \{\text{meth } m_1 = a_i; \ldots \text{meth } m_q = a_i\}$$

Given arguments for parameter $x_1, \ldots x_n$, it builds recursive methods $m_1, \ldots m_n$. A class can be compiled into an object closure:

$$\begin{array}{l} \text{let } m = \\ \qquad \{ \quad m_1 = \lambda(m, x_1, \ldots x_q).\, [\![a_1]\!]; \\ \qquad \qquad \vdots \\ \qquad \quad\ m_p = \lambda(m, x_1, \ldots x_q).\, [\![a_p]\!] \quad \} \text{ in} \\ \lambda x_1, \ldots x_q.\, (m, x_1, \ldots x_q) \end{array}$$

Each m_i is bound to the code for the corresponding method. All codes are combined into a record of codes. Then, calling method m_i of an object p is $(\mathsf{proj}_0\, p).m_i\, p$.

Let us write the typed version of this encoding. Let τ_i be the type of M_i and row R describe the types of $(x_1, \ldots x_q)$. Let $\mathsf{Clo}(R)$ be $\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i \in 1..n}\}; R)$ and $\mathsf{UClo}(R)$ its unfolding.

Fields R are hidden in an existential type $\mu\alpha.\Pi(\{(m_i : \alpha \to \tau_i)^{i \in I}\}; \rho)$:

$$\begin{array}{l} \mathsf{let}\ m = \\ \quad \{ \quad m_1 = \lambda(m, x_1, \ldots x_q : \mathsf{UClo}(R)).\, [\![M_1]\!]; \\ \qquad \vdots \\ \qquad m_p = \lambda(m, x_1, \ldots x_q : \mathsf{UClo}(R)).\, [\![M_p]\!] \quad \}\ \mathsf{in} \\ \lambda x_1. \ldots \lambda x_q.\, \mathsf{pack}\ R, \mathsf{fold}\ (m, x_1, \ldots\, x_q)\ \mathsf{as}\ \exists \rho.\, (M, \rho) \end{array}$$

Calling a method of an object p of type M is

$$p\#m_i \ \triangleq\ \mathsf{let}\ \rho, z = \mathsf{unpack}\ p\ \mathsf{in}\ (\mathsf{proj}_0\ \mathsf{unfold}\ z).m_i\ z$$

An object has a recursive type but it is *not* a recursive value.

Typed encoding of objects were first studied in the 90's to understand what objects really are in a type setting. These encodings are in fact type-preserving compilation of (primitive) objects. There are several variations on these encodings. See Bruce et al. (1999) for a comparison. See Rémy (1994a) for an encoding of objects in (a small extension of) ML with iso-existentials and universals. See Abadi and Cardelli (1996, 1995) for more details on primitive objects.

Summary

Type-preserving compilation is rather *fun.* (Yes, really!) It forces compiler writers to make the structure of the compiled program *fully explicit,* in type-theoretic terms. In practice, building explicit type derivations, ensuring that they remain small and can be efficiently typechecked, can be a lot of work.

Because we have focused on type preservation, we have studied only naive closure conversion algorithms. More ambitious versions of closure conversion require program analysis: see, for instance, Steckler and Wand 1997. These versions *can* be made type-preserving.

Defunctionalization, an alternative to closure conversion, offers an interesting challenge, with a simple solution. See, for instance Pottier and Gauthier (2006). Designing an efficient, type-preserving compiler for an *object-oriented language* is quite challenging. See, for instance, Chen and Tarditi (2005).

One may think that references in System F could be translated away by making the store explicit. In fact, this can be done, but not in System F, nor even in System F^ω: the translation is quite tricky and in order for the translation to be well-typed the type system must be reach enough to express monotonicity of the store in a context where the store is itself recursively defined. See Pottier (2011) for details.

Exercise 47 (CPS conversion) *Here is an untyped version of call-by-value CPS conversion:*

$$
\begin{aligned}
[\![V]\!] &= \lambda k.\, k\, (\!|V|\!) \\
[\![M_1\, M_2]\!] &= \lambda k.\, [\![M_1]\!]\, (\lambda x_1.\, [\![M_2]\!]\, (\lambda x_2.\, x_1\, x_2\, k))
\end{aligned}
\qquad
\begin{aligned}
(\!|x|\!) &= x \\
(\!|()|\!) &= () \\
(\!|(V_1, V_2)|\!) &= ((\!|V_1|\!), (\!|V_2|\!)) \\
(\!|\lambda x.\, M|\!) &= \lambda x.\, [\![M]\!]
\end{aligned}
$$

Is this a type-preserving transformation? *(Solution p. 169)* □

Chapter 7

Overloading

7.1 An overview

Overloading occurs when several definitions of an identifier may be visible simultaneously at the same occurrence in a program. An interpretation of the program (and a fortiori a run of the program) must choose the definition that applies at this occurrence. This is called overloading *resolution*. Overloading resolution may use quite different strategies and techniques. All sorts of identifiers may be subject to overloading: variables, labels, constructors, types, etc.

Overloading must be distinguished from shadowing of identifiers by normal scoping rules, where in this case, a definition is just temporarily inaccessible by another one, but only the last definition is visible.

7.1.1 Why use overloading?

There are several reasons to use overloading.

Overloading may just be a naming convenience that allows reusing the same identifier for similar but different operations. This avoids name mangling such as suffixing similar names by type information: printing functions, *e.g.* *print_int*, *print_string*, *etc.*; numerical operations, *e.g.* (+), .+ *etc.*); or numerical constants *e.g.* 0, 0., *etc.* In this respect, it may help with modularity. In the absence of overloading, the naming discipline (including name mangling conventions) must be known *globally* to avoid name clashes, which breaks compositionality. Isolated identifiers with no particular naming convention may still interfere between different developments and cannot be used together unless fully qualified. This problem does not disappear with overloading but it may be minimized—as long as overloading is not ambiguous. Hence, in some sense, overloading allows to think more abstractly, in terms of operations rather than of particular implementations. For instance, calling *to_string* conversion lets the system check whether one definition is available according to the type of the argument.

Overloading definitions may also be used to provided type dependent functions. That is, a function may be defined for all types $\tau[\alpha]$ but with an implementation depending on the type of α by provided several overloaded definitions for different types $\tau[\tau_i]$. For instance, a marshaling function of type $\forall\alpha.\,\alpha \to \mathsf{string}$ may execute different code for each base type α.

Overloading definitions may be *ad hoc*, *i.e.* completely unrelated for each type—or just share a same type schema. For example 0 could mean either the integer zero or the empty list; and "×" could mean either the integer product or string concatenation.

Conversely, overloaded definitions may depend solely on the *type structure* (*i.e.* on whether the argument is a sum, a product, *etc.*) so that definitions can be derived mechanically for all types from their definitions on base types. Such overloaded functions are called polytypic functions. Typical examples are marshaling functions, or the generation of random values for arbitrary types as used in the Quickcheck tool for Haskell. *etc.* Still, polytypic definition often need to be specialize at some particular types. For example, one may use a polytypical definition of printing, so that printing is available at all types, but define specialized versions of printing at some particular types.

7.1.2 Different forms of overloading

There are many variants of overloading. They can be classified by how overloading is *introduced* and *resolved*.

The first elements of classification are the restrictions on overloading definitions. Can arbitrary definitions be overloaded? For instance, can numerical values be overloaded? Are all overloaded definitions of the same symbol instances of a common type scheme? Are these type schemes arbitrary? Are overloaded definitions primitive (pre-existing), automatic (generated mechanically from other definitions), or user-defined? Can overloaded definitions overlap? Can overloaded definitions have a local scope?

However, the main element of classification remains the resolution strategy—which may indirectly constraint the way overloading is introduced. We distinguish between *static* and *dynamic* resolutions strategies.

Static resolution of overloading has a very simple semantics since the meaning of the program can be determined statically by deciding for each overloaded symbol which actual definition of the symbol should be used. Hence, it replaces each occurrence of an overloaded symbol by an actual implementation at the appropriate type. Therefore static overloading does not increase expressiveness per say, since the user could have chosen the appropriate implementation in the first place. Still, static overloading may significantly reduce verbosity—and increase modularity and abstraction, as explained above.

Conversely, dynamic resolution increrases expressiveness, as the choice of the implementation may now depend on the dynamic of the program execution. However, it is also much more involved, since the semantics of the language usually need extra machinary to support the dynamic resolution. For example, the resolution of some occurrence of a polymorphic

function may depend on the type of its arguments, so that different calls of the function at different types can make different choices. The resolution is driven by information made available at runtime: it could at worse require full type information. In some restrictions, partial type information may be sufficient, and sometimes some type-related information can be used instead of types themselves, such as tags, dictionaries, *etc.* These can be attached to values (as tags in object oriented languages), or passed as extra arguments at runtime (as dictionaries in Haskell).

7.1.3 Static overloading

The language SML has a very limited form of overloading where overloaded definitions are primitive: they include an exhaustive list of overloaded definitions for numerical operators, plus automatically generated overloaded definitions for all record accessors. The resolution is static and fails if overloading cannot be unambiguously resolved at outermost let-definitions. For example, `let` $\mathit{twice}\ x = x + x$ is rejected in SML at toplevel, since + could be either the addition on either integers or floats.

In the language Java, overloading is not primitive but automatically generated by subtyping: when a class extends another one and a method is redefined, the older definition is still visible, but at another type, hence the method is overloaded. This overloading is then statically resolved by choosing the most specific definition. There is always a best choice—according to static knowledge. This static resolution of overloading in Java comes in complement to the dynamic dispatch of method calls. This is often a source of confusion for programmers who often expect a dynamic resolution of overloading and as a result misunderstand the semantics of their programs. For instance, an argument may have a runtime type that is a subtype of the best known compile-time type, and perhaps a more specific definition could have been used if overloading were resolved dynamically.

However convenient, static resolution of overloading is quite limited. Moreover, it does not fit very well with first-class functions and polymorphism. Indeed, with static overloading, $\lambda x.\, x + x$ is rejected when + is overloaded, as it cannot be resolved. The function must be manually specialized at some type for which + is defined. This argues in favor of some form of dynamic overloading that allows to delay resolution of overloaded symbols at least until polymorphic functions have been sufficiently specialized.

7.1.4 Dynamic resolution with a type passing semantics

The most ambitious approach to dynamic overloading is to pass types at runtime and dispatch on the runtime type, using a general typecase construct.

Runtime type dispatch is the most general approach as it does not impose much restriction on the introduction of overloaded definitions It uses an explicitly-typed calculus (*e.g.* System F)—with a type passing semantics—extended with a typecase construct. However,

the runtime cost of typecase may be high, unless type patterns are significantly restricted. Moreover, one pays even when overloading is not used, since types are always passed around, even when overloading is not used, unless the compiler uses aggressive program analyzes to detech these sitiations and optimize type computations away. Monomorphization may also be used to allow more static resolution in such cases. Ensuring exhaustiveness of type matching is often a difficult task in this context.

The ML& calculus by Castagna (1997) offers a general overloading mechanism based on type dispatch. It is an extension of System F with intersection types, subtyping, and type matching. An expressive type system keeps track of exhaustiveness; type matching functions are first-class and can be extended or overridden. The language allows overlapping definitions with a best match resolution strategy.

7.1.5 Dynamic overloading with a type erasing semantics

To avoid the expensive cost of typecase, one may restrict the overloaded definitions, so that full type information is not needed and only an approximation of types, such as tags, may be used for overloading resolution. This is one possible approach to object-orientation in the *method as overloading functions* paradigm where object classes are used to dynamically select the appropriate method. This is also an approach used in some scheme dialects known as *generics*.

In fact, one may get more freedom by detaching tags from values and passing tags—or almost equivalently passing the actually implementations grouped into dictionaries—as extra runtime arguments. A side advantage of this approach is that the semantics can be described without changing the runtime environment, *i.e.* the representation of values, as an elaboration process that introduces abstractions and applications for implementations of overloaded symbols. Schematically, one transforms unresolved overloaded symbols into extra abstractions and passes actual implementations (or abstractions of implementations) around as extra arguments. Hopefully, overloaded symbols can be resoled when their types are sufficiently specialized and before they are actually needed.

For example, a program context $\mathsf{let}\ f = \lambda x.\, x + x\ \mathsf{in}\ []$ can be elaborated into $\mathsf{let}\ f = \lambda(+).\, \lambda x.\, x + x\ \mathsf{in}\ []$. If $f\ 1.0$ is placed in the hole of this original program context, it can then be elaborated to $f\ (+.)\ 1.0$, which can be placed in the hole of the elaborated program context. Elaboration can be performed after typechecking by translating the typing derivation. After elaboration, types are no longer needed and can be erased. Monomorphization or other simplifications may reduce the number of abstractions and applications introduced by overloading resolution.

This technique has been widely explored—under different facets—in the context of ML: Type classes, introduced very early by Wadler and Blott (1989) are still the most popular and widely used framework. Other contemporary solutions have been proposed by Rouaix (1990) and Kaes (1992). Simplifications of type classes have also been proposed by Odersky et al.

(1995) but did not take over, because of their restrictions. Recent works on type classes is still going on Morris and Jones (2010).

In the rest of this chapter we introduce a tiny language called Mini Haskell that models the essence of Haskell type classes; at the end we also discuss *implicit arguments* as a less structured but simpler way of introducing dynamic overloading in a programming language.

7.2 Mini Haskell

Mini Haskell—or MH for short—is a simplification of Haskell to avoid most of the difficulties of type classes but keeping their essence: it is restricted to single parameter type classes and no overlapping instance definitions; it is close in expressiveness and simplicity to *A second look at overloading* by Odersky et al. but closer to Haskell in style—it can be easily generalized by lifting restrictions without changing the framework.

The language MH is explicitly typed. In this section, we first present some examples in MH, and then describe the language and its elaboration into System F. We introduce an implicitly-typed version of MH and its elaboration in the next section.

7.2.1 Examples in MH

An equality class and several instances many be defined in Mini Haskell as follows:

$$
\begin{array}{l}
\texttt{class}\ \mathit{Eq}\ (X)\quad \{\ \mathrm{equal} : X \to X \to \mathit{Bool}\ \} \\
\texttt{inst}\ \ \mathit{Eq}\ (\mathit{Int})\ \ \{\ \mathrm{equal} = \mathit{primEqInt}\ \} \\
\texttt{inst}\ \ \mathit{Eq}\ (\mathit{Char})\ \{\ \mathrm{equal} = \mathit{primEqChar}\ \} \\
\texttt{inst}\ \ \Lambda(X)\ \mathit{Eq}\ (X) \Rightarrow \mathit{Eq}\ (\mathit{List}\ (X)) \\
\quad \{\ \mathrm{equal} = \lambda(l_1 : \mathit{List}\ X)\ \lambda(l_2 : \mathit{List}\ X)\ \texttt{match}\ l_1,\ l_2\ \texttt{with} \\
\qquad\qquad \mid\ [],[]\ \to \mathit{true}\ \mid\ [],\_\ \mid\ [],\_\ \to \mathit{false} \\
\qquad\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to \mathrm{equal}\ X\ h_1\ h_2\ \&\&\ \mathrm{equal}\ (\mathit{List}\ X)\ t_1\ t_2\ \}
\end{array}
$$

This code declares a class (dictionary) of type Eq(X) that contains definitions for $\mathrm{equal} : X \to X \to X$ and creates two concrete instances (dictionaries) of type $\mathit{Eq}(\mathit{Int})$ and $\mathit{Eq}(\mathit{Char})$, and a function that, given a dictionary for $\mathit{Eq}(X)$, builds a dictionary for type $\mathit{List}(X)$. This code can be elaborated by explicitly building dictionaries as records of functions:

$$
\begin{array}{l}
\texttt{type}\ \mathit{Eq}\ (X) = \{\ \mathrm{equal} : X \to X \to \mathit{Bool}\ \} \\
\texttt{let}\ \mathrm{equal}\ X\ (\mathrm{EqX} : \mathit{Eq}\ X) : X \to X \to \mathit{Bool} = \mathrm{EqX}.\mathrm{equal} \\
\\
\texttt{let}\ \mathrm{EqInt} : \mathit{Eq}\ \mathit{Int}\ \ = \{\ \mathrm{equal} = (\ \mathit{primEqInt} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\ )\ \} \\
\texttt{let}\ \mathrm{EqChar} : \mathit{Eq}\ \mathit{Char} = \{\ \mathrm{equal} = \mathit{primEqChar}\ \} \\
\\
\texttt{let}\ \mathrm{EqList}\ X\ (\mathrm{EqX} : \mathit{Eq}\ X) : \mathit{Eq}\ (\mathit{List}\ X) = \\
\quad \{\ \mathrm{equal} = \lambda(l_1 : \mathit{List}\ X)\ \lambda(l_2 : \mathit{List}\ X)\ \texttt{match}\ l_1,\ l_2\ \texttt{with} \\
\qquad \mid\ [],[]\ \to \mathit{true}\ \mid\ [],\_\ \mid\ [],\_\ \to \mathit{false} \\
\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to
\end{array}
$$

$$\mathrm{equal}\ X\ \mathrm{EqX}\ h_1\ h_2\ \&\&\ \mathrm{equal}\ (\mathit{List}\ X)\ (\mathrm{EqList}\ X\ \mathrm{EqX})\ t_1\ t_2\ \}$$

Classes may themselves depend on other classes (called superclasses), which realizes a form of class inheritance.

$$\begin{array}{l} \texttt{class}\ \mathit{Eq}\ (X) \Rightarrow \mathit{Ord}\ (X)\ \{\ \mathrm{lt} : X \to X \to \mathit{Bool}\ \} \\ \texttt{inst}\ \mathit{Ord}\ (\mathit{Int})\ \{\ \mathrm{lt} = (<)\ \} \end{array}$$

The class definition declares a new class (dictionary) $\mathit{Ord}\ (X)$ that contains a method $\mathit{Ord}(X)$ that depends on a dictionary $\mathit{Eq}(X)$ and contains a method $\mathrm{lt} : X \to X \to \mathit{Bool}$. The instance definition builds a dictionary $\mathit{Ord}(\mathit{Int})$ from the existing dictionary $\mathit{Eq}\ \mathit{Int}$ and the primitive $(<)$ for lt. The two declarations are elaborated into:

$$\begin{array}{l} \texttt{type}\ \mathit{Ord}\ (X) = \{\ \mathit{Eq} : \mathit{Eq}\ (X);\ \mathrm{lt} : X \to X \to \mathit{Bool}\ \} \\ \texttt{let}\ \mathrm{EqOrd}\ X\ (\mathrm{OrdX} : \mathit{Ord}\ X) : \mathit{Eq}\ X = \mathrm{OrdX}.\mathit{Eq} \\ \texttt{let}\ \mathrm{lt}\ X\ (\mathrm{OrdX} : \mathit{Ord}\ X) : X \to X \to \mathit{Bool} = \mathrm{OrdX}.\mathrm{lt} \\ \texttt{let}\ \mathrm{OrdInt} : \mathit{Ord}\ \mathit{Int} = \{\ \mathit{Eq} = \mathrm{EqInt};\ \mathrm{lt} = (<)\ \} \end{array}$$

So far, we have just defined type classes and some instances. We may write a function that uses these overloaded definitions. When overloading cannot be resolved statically, the function will be abstracted other one or several additional arguments, called dictionnaries, that will carry the appropriate definitions for the unresolved overloaded symbols. For example, consider the following definition in Mini Haskell:

$$\begin{array}{l} \texttt{let rec}\ \mathit{search} : \forall(X)\ \mathit{Ord}\ X \Rightarrow X \to \mathit{List}\ X \to \mathit{Bool} = \\ \quad \Lambda(X)\ \lambda(x : X)\ \lambda(l : \mathit{List}\ X) \\ \quad\quad \texttt{match}\ l\ \texttt{with}\ [\,] \to \mathit{false} \mid h{::}t \to \mathrm{equal}\ x\ h\ \|\ \mathit{search}\ X\ x\ t \end{array}$$

This code is elaborated into:

$$\begin{array}{l} \texttt{let rec}\ \mathit{search}\ X\ (\mathrm{OrdX} : \mathit{Ord}\ X)\ (x : X)\ (l : \mathit{List}\ X) : \mathit{Bool} = \\ \quad \texttt{match}\ l\ \texttt{with}\ [\,] \to \mathit{false} \\ \quad\quad \mid h{::}t \to \mathrm{equal}\ X\ (\mathrm{EqOrd}\ X\ \mathrm{OrdX})\ x\ h\ \|\ \mathit{search}\ X\ \mathrm{OrdX}\ x\ t \end{array}$$

Using the overloading function, as in $\mathit{search}\ \mathit{Int}\ 1\ [1;\ 2;\ 3]$ will then elaborate into the code $\mathit{search}\ \mathit{Int}\ \mathrm{OrdInt}\ 1\ [1;\ 2;\ 3]$ where a dictionary OrdInt of the appropriate type has been built and passed as an additional argument. Here, the target language is the explicitly-typed System F, which has a type erasing semantics, hence the type argument Int may be dropped while the dictionary argument OrdInt is retained: the code that is actually executed is thus $\mathit{search}\ \mathrm{OrdInt}\ 1\ [1;\ 2;\ 3]$ (where type information has been stripped off OrdInt itself).

7.2.2 The definition of Mini Haskell

Class declarations and instance definitions are restricted to the toplevel. Their scope is the whole program. In practice, a program p is a sequence of class declarations and instance and function definitions given in any order and ending with an expression. For simplification,

$$
\begin{array}{rcl}
p & ::= & H_1 \dots H_p \; h_1 \dots h_q \; M \\
H & ::= & \mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\} \\
\rho & ::= & u_1 : \tau_1, \dots u_n : \tau_n \\
h & ::= & \mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\} \\
r & ::= & u_1 : M_1, \dots u_n : M_n
\end{array}
\qquad
\begin{array}{rcl}
P & ::= & \mathsf{K}\ \alpha \\
\vec{P} & ::= & P_1, \dots P_n \\
Q & ::= & \mathsf{K}\ \tau \\
\vec{Q} & ::= & Q_1, \dots Q_n \\
\sigma & ::= & \forall \vec{\alpha}. \vec{Q} \Rightarrow T \\
T & ::= & \tau \mid Q
\end{array}
$$

Figure 7.1: Syntax of MH expressions and types

we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

Instance definitions are interpreted recursively and their definition order does not matter. We may assume, *w.l.o.g.*, that instance definitions come after all class declarations. The order of class declaration matters, since they may only refer to other class constructors that have been previously defined.

For sake of simplification, we restrict to single parameter classes. The syntax of MH programs is defined in Figure 7.1. Letter p ranges over source programs. A program p is a sequence $H_1 \dots H_p\ h_1 \dots h_q\ M$, of class declaration $H_1 \dots H_p$, followed by a sequence of instance definitions $h_1 \dots h_q$, and ending with an expression M.

A class declaration H is of the form $\mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$. It defines a new class (constructor) K, parametrized by α. Every class (constructor) K must be defined by one and only one class declaration. So we may say that H is the declaration of K and write H_{K}.

Letter u ranges over *overloaded symbols*, also called *methods*. The row ρ of the form $u_1 : \tau_1, \dots u_n : \tau_n$ declares overloaded symbols u_i of class K. An overloaded symbol cannot be declared twice in a program; it cannot be repeated twice in the same class (hence the map $i \mapsto u_i$ is injective) and cannot be declared in two different classes. The row ρ (and thus each of its field type τ_i) must not contain any other free variable than α.

The class depends on a sequence of subclasses $\vec{P}$ of the form $\mathsf{K}_1\ \alpha, \dots \mathsf{K}_n\ \alpha$, which is called a *typing context*. Each clause $\mathsf{K}_i\ \alpha$ can be read as an assumption *"given an instance of class* K_1 *at type* α*"* and $\vec{P}$ as the conjunction of these assumptions. We say that classes K_i's are superclasses of K which we write $\mathsf{K}_i \prec \mathsf{K}$. They must have been previously defined. This ensures that the relation $\prec$ is acyclic. We require that all K_i's are independent, *i.e.* there doa not exists i and j such that $\mathsf{K}_j \prec \mathsf{K}_i$.

An instance definition h is of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$. It defines an instance of a class K at type $\mathsf{G}\ \vec{\beta}$ where G is a datatype constructor, *i.e.* neither an arrow type nor a class constructor. A class constructor K may appear in Q but not in τ. An instance definition *d*efines the methods of a class at the required type: r is a record of methods

$u_1 = M_1, \ldots u_n = M_n$.

An instance definition is also parametrized by a typing context $\vec{P}$ of the form $K_1\ \alpha_1, \ldots K_k\ \alpha_k$ where variables α_i's are included in $\vec{\beta}$. This typing context is is not related to the typing context of its class declaration H_{K}, but to the set of classes that the implementations of the methods depend on.

Restrictions The restriction to types of the form $\mathsf{K}'\ \alpha'$ in typing contexts and class declarations, and to types of the form $\mathsf{K}'\ (\mathsf{G}'\ \vec{\alpha}')$ in instances are for simplicity. Generalization are possible and discussed later (§7.4).

7.2.3 Semantics of Mini Haskell

The semantics of Mini Haskell is given by elaborating source programs into System F extended with record types and recursive definitions. Record types are provided as data types. They are used to represent dictionaries. Record labels are used to encode overloaded identifiers u. We may use overloaded symbols as variables as well: this amounts to reserving a subset of variables x_u indexed by overloaded symbols and writing u as a shortcut for x_u. We use letter N instead of M for elaborated terms, to distinguish them from source terms. For convenience, we write $\Rightarrow$ in System F as an alias for $\rightarrow$, which we use when the argument is a (record representing a) dictionary. Type schemes in the target language take the form σ described on Figure 7.1. Notice that types T are stratified: they are either dictionary types $\mathsf{K}\ \tau$ or a regular type τ that does not contain dictionary types.

Class declaration The elaboration of a class declaration H_{K} of the form $\mathsf{class}\ \mathsf{K}_1\ \alpha, \ldots \mathsf{K}_n\ \alpha \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$ consists of several parts. It first declares a record type that will be used as a dictionary to carry both the methods and the dictionaries of its *immediate* superclasses. A class need not contain subdictionaries recursively, since if $\mathsf{K}_j \prec \mathsf{K}_i$, then a dictionary for K_i already contains a sub-dictionary for K_j, to which K has access via K_i so it does need not have one itself. The row ρ of the class definition only lists the class methods. Hence, we extend it with fields for sub-dictionaries and define the record type:

$$\mathsf{K}\ \alpha \approx \{\rho^{\mathsf{K}}\} \qquad \text{where } \rho^{\mathsf{K}} \text{ is } u^{\mathsf{K}}_{\mathsf{K}_1} : \mathsf{K}_1\ \alpha, \ldots u^{\mathsf{K}}_{\mathsf{K}_n} : \mathsf{K}_n\ \alpha, \rho.$$

This record type declaration is collected to appear in the program *prelude*.

Then, for each $u : T_u$ in ρ^{K}, we define the program context:

$$\mathcal{R}_u \triangleq \mathsf{let}\ u : \sigma_u = N_u\ \mathsf{in}\ [\,] \qquad \text{where} \quad \sigma_u \triangleq \forall \alpha.\, \mathsf{K}\ \alpha \Rightarrow T_u \ \text{ and } \ N_u \triangleq \Lambda\alpha.\lambda z{:}\mathsf{K}\ \alpha.\,(z.u)$$

Let the composition $\mathcal{R}_1 \circ \mathcal{R}_2$ of two contexts be the context $\mathcal{R}_1[\mathcal{R}_2]$ obtained by placing $\mathcal{R}_2$ in the hole of $\mathcal{R}_1$. The elaboration $[\![H_{\mathsf{K}}]\!]$ of a single class declaration H_{K} is the composition:

$$[\![H_{\mathsf{K}}]\!] \triangleq \mathcal{R}_{u_1} \circ \ldots \mathcal{R}_{u_n} \qquad \text{where} \quad \mathsf{K}\ \alpha \approx \{u_1 : T_1, \ldots u_n : T_n\}$$

that defines accessors for each field of the class dictionary. We also define the typing environment Γ_H as an abbreviation for $u_1 : \sigma_{u_1}, \ldots u_n : \sigma_{u_n}$.

The elaboration $[\![H_1 \ldots H_p]\!]$ of all class definitions is the composition $[\![H_1]\!] \circ \ldots [\![H_p]\!]$ of the elaboration of each. We also define $\Gamma_{H_1 \ldots H_n}$ as the concatenation $\Gamma_{H_1}, \ldots \Gamma_{H_n}$ of individual typing environments.

Instance definition In an instance declaration h of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$, The typing context $\vec{P}$ describes the dictionaries that must be available on type parameters $\vec{\beta}$ for constructing the dictionary $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, but that cannot yet be built because they depend on some unknown type β in $\vec{\beta}$.

As mentioned above $\vec{P}$ is not related to the typing context of the class declaration H_{K}. To see this, assume that class K' is an immediate superclass of K, so that the creation of the dictionary $\mathsf{K}\ \alpha$ requires the existence of a dictionary $\mathsf{K}'\ \alpha$; then, an instance declaration $\mathsf{K}\ \mathsf{G}$ (where G is nullary) need not be parametrized over a dictionary of type $\mathsf{K}'\ \mathsf{G}$, as either such a dictionary can already be built, hence the instance definition does not require it, or it will never be possible to build one, as instance definitions are recursively defined so all of them are already visible—and the program must be rejected.

We restrict typing context $\mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k$ to *canonical* ones defined as satisfying the two following conditions: (1) α_i is some β_j in $\vec{\beta}$; and (2) if K_i and K_j are related, *i.e.* $\mathsf{K}_i \prec \mathsf{K}_j$ or $\mathsf{K}_j \prec \mathsf{K}_i$ or $\mathsf{K}_i = \mathsf{K}_j$. then α_i and α_j are different. The latter condition avoids having two dictionaries $\mathsf{K}_i\ \beta$ and $\mathsf{K}_j\ \beta$ when, *e.g.*, $\mathsf{K}_i \prec \mathsf{K}_j$ since the former is contained in the latter.

The elaboration of an instance declaration h is a triple (z_h, N^h, σ_h) where z_h is an identifier to refer to the elaborated body N^h of type

$$\sigma_h \ \triangleq\ \forall \beta_1 \ldots \beta_p.\, \mathsf{K}_1\ \alpha_1 \Rightarrow \ldots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$$

(Variables $\alpha_1, \ldots \alpha_k$ are among $\beta_1, \ldots \beta_p$ and may contain repetitions, as explained above.) The expression N^h builds a dictionary of type $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, given k dictionaries (where k may be *zero*) of respective types $\mathsf{K}_1\ \beta_1, \ldots \mathsf{K}_k\ \beta_k$ and is defined as:

$$\begin{array}{rcl} N^h & \triangleq & \Lambda\beta_1.\ \ldots \Lambda\beta_p.\ \lambda(z_1 : \mathsf{K}_1\ \alpha_1).\ \ldots \lambda(z_k : \mathsf{K}_k\ \alpha_k). \\ & & \qquad \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N^h_1, \ldots u_m = N^h_m\} \end{array}$$

The types of fields are as prescribed by the class definition K, but specialized at type $\mathsf{G}\ \vec{\beta}$. That is, q_i is a dictionary expression of type $\mathsf{K}'_i\ (\mathsf{G}\ \vec{\beta})$ whose exact definition is postponed until the elaboration of dictionaries in §7.2.6. The term N^h_i is the elaboration of M_i where $u_1 = M_1, \ldots u_m = M_m$ is r; it is described in the next section (§7.2.4). For clarity, we write z instead of x when a variable binds a dictionary or a function building a dictionary. Notice that the expressions q_i and N^h_i sees the type variables $\beta_1, \ldots \beta_p$ and the dictionary parameters $z_1 : \mathsf{K}_1\ \alpha_1, \ldots z_k : \mathsf{K}_k\ \alpha_k$.

The elaboration of all instance definitions is the program context:

$$[\![\vec{h}]\!] \triangleq \mathsf{let\ rec}\ (\vec{z}_h : \vec{\sigma}_h) = \vec{N}^h\ \mathsf{in}\ []$$

that recursively binds all instance definitions in the hole.

Program Finally, the elaboration of a complete program $\vec{H}\ \vec{h}\ M$ is

$$[\![\vec{H}\ \vec{h}\ M]\!] \triangleq ([\![\vec{H}]\!] \circ [\![\vec{h}]\!])[M] = \mathsf{let}\ \vec{u} : \vec{\sigma}_u = \vec{N}_u\ \mathsf{in\ let\ rec}\ (\vec{z}_h : \vec{\sigma}_h) = \vec{N}^h\ \mathsf{in}\ N$$

Hence, the expression N, which is the elaboration of M, and all expressions N_h are typed (and elaborated) in the environment $\Gamma_{\vec{H}\vec{h}}$ equal to $\Gamma_{\vec{H}}, \Gamma_{\vec{h}}$: the environment $\Gamma_{\vec{H}}$ declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols) while the environment $\Gamma_{\vec{h}}$, declares functions to build dictionaries.

7.2.4 Elaboration of expressions

The elaboration of expressions is defined by a judgment $\Gamma \vdash M \rightsquigarrow N : \sigma$ where Γ is a System F typing context, M is the source expression, N is the elaborated expression and σ its type in Γ. In particular, $\Gamma \vdash M \rightsquigarrow N : \sigma$ implies $\Gamma \vdash N : \sigma$ in System F.

We write q for dictionary terms, which are the following subset of System-F terms:

$$q ::= u \mid z \mid q\ \tau \mid q\ q$$

Variables u and z are just particular cases of variables x. Variable u is used for methods (and access to subdictionaries), while variable z is used for dictionary parameters and for class instances, *i.e.* dictionaries or functions building dictionaries.

The rules for elaboration of expressions are described in Figure 7.2. Most of them just wrap the elaboration of their sub-expressions. In rule Let, we require σ to be canonical, *i.e.* of the form $\forall\vec{\alpha}.\vec{P} \Rightarrow T$ where $\vec{P}$ is itself empty or canonical (see page 153). Rules App and Abs do not apply to overloaded expressions of type σ but only to simple expressions of type τ.

The interesting rules are the elaboration of overloaded expressions, and in particular of missing abstractions (Rule OAbs) and applications (Rule OApp) of dictionaries. Rule OAbs pushes dictionary abstractions in the context Γ as prescribed by the expected type. On the opposite, Rule OApp searches for an appropriate dictionary-building function and applies it to the required sub-directionary.

The premise $\Gamma \vdash q : Q$ of rule OApp also triggers the elaboration of dictionaries. This judgment is just the typability in System F—but restricted to dictionary expressions. That is, it searches for a well-typed dictionary expression. The restriction to dictionary expressions ensures that under reasonable conditions the search is decidable—and coherent. The elaboration of dictionaries reads the typing rules of System F restricted to dictionaries as an algorithm, where Γ and Q are given and q is inferred. This is described in detail in §7.2.6.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \sigma}\ \textsc{Var} \qquad \frac{\Gamma \vdash M \rightsquigarrow N : \forall \alpha.\, \sigma}{\Gamma \vdash M\ \tau \rightsquigarrow N\ \tau : [\alpha \mapsto \tau]\sigma}\ \textsc{Inst} \qquad \frac{\Gamma, \alpha \vdash M \rightsquigarrow N : \sigma}{\Gamma \vdash \Lambda\alpha.M \rightsquigarrow \Lambda\alpha.N : \forall \alpha.\, \sigma}\ \textsc{Gen}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 \rightsquigarrow N_2 : \tau}{\Gamma \vdash \mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2 \rightsquigarrow \mathsf{let}\ x : \sigma = N_1\ \mathsf{in}\ N_2 : \tau}\ \textsc{Let} \qquad \frac{\Gamma, x : \tau' \vdash M \rightsquigarrow N : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, M \rightsquigarrow \lambda x{:}\tau'.\, N : \tau' \to \tau}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 \rightsquigarrow N_2 : \tau_2}{\Gamma \vdash M_1\ M_2 \rightsquigarrow N_1\ N_2 : \tau_1}\ \textsc{App} \qquad \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda x{:}Q.\, N : Q \Rightarrow \sigma}\ \textsc{OAbs}$$

$$\frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}\ \textsc{OApp}$$

Figure 7.2: Elaboration of expressions

By construction, elaboration produces well-typed expressions: that is $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \tau$ implies that is $\Gamma_{\vec{H}\vec{h}} \vdash N : \tau$.

7.2.5 Summary of the elaboration

An instance declaration h of the form:

$$\mathsf{inst}\ \forall \vec{\beta}.\ \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ \vec{\tau}\ \{u_1 = M_1, \ldots l; u_m = M_m\}$$

is translated into

$$\lambda(z_1{:}\mathsf{K}_1\ \alpha_1) \ldots .\, \lambda(z_p{:}\mathsf{K}_k\ \alpha_k).\, \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots\ u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N_1, \ldots\ u_m = N_m\}$$

where $u^{\mathsf{K}}_{\mathsf{K}'_i} : \tau_i$ are the superclasses fields, Γ^h is $\vec{\beta}, \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k$, and the following elaboration judgments $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash q_i : \tau_i$ and $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash M_i \rightsquigarrow N_i : \tau_i$ hold. Finally, given the program p equal to $\vec{H}\ \vec{h}\ M$, we elaborate M as N such that $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \forall \bar{\alpha}.\, \tau$.

Notice that $\forall \bar{\alpha}.\, \tau$ is an unconstrained type scheme. Otherwise, N could elaborate into an abstraction over dictionaries, which could turn a computation into a function that is not reduced: this would not preserve the intended semantics.

More generally, we must be careful to preserve the *intended* semantics of source programs. For this reason, in a call-by-value setting, we must not elaborate applications into abstractions, since this could delay and perhaps duplicate the order of evaluations. We just pick the obvious solution, that is to restrict rule Let so that either σ is of the form $\forall \bar{\alpha}.\, \tau$ or M_1 is a value or a variable.

In a language with a call-by-name semantics, an application is not evaluated until it is needed. Hence adding an abstraction in front of an application should not change the evaluation order $M_1\ M_2$. We must in fact compare:

$$\mathsf{let}\ x_1 = \lambda y.\,\mathsf{let}\ x_2 = V_1\ V_2\ \mathsf{in}\ M_2\ \mathsf{in}\ [x_1 \mapsto x_1\ q]M_1 \qquad \textbf{(1)}$$
$$\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \lambda y.\,V_1\ V_2\ \mathsf{in}\ [x_2 \mapsto x_2\ q]M_2\ \mathsf{in}\ M_1 \qquad \textbf{(2)}$$

The order of evaluation of $V_1\ V_2$ is preserved. However, the Haskell language is call-by-need and not call-by-name! Hence, applications are delayed as in call-by-name but shared and only reduced once. The application $V_1\ V_2$ will be reduced once in (1), but as many types as there are occurrences of x_2 in M_2 in (2).

The final result will still be the same in both cases if the language has no side effects, but the intended semantics may be changed regarding the complexity.

Coherence The elaboration may fail for several reasons: The input expression may not obey one of the restrictions we have requested; a typing may occur during elaboration of an expression; or or some dictionary cannot be build. If elaboration fails, the program p is rejected, of course.

When the elaboration of p succeeds, it should return a term $[\![p]\!]$ that is well-typed in F and that defines the semantics of p. However, although terms are explicitly-typed, their elaboration may not be unique! Indeed, they might be several ways to build dictionaries of some given type, as we shall see below (§7.2.6).

We may distinguish two situations: in the worst case, a source program may elaborate to several completely unrelated programs; in the better case, all possible elaborations may in fact be *equivalent* programs: we say that the elaboration is *coherent* and the programs has a deterministic semantics given by any of its elaboration.

Opening a parenthesis, what does it mean for programs be equivalent? There are several notions of program equivalence:

- If programs have a denotational semantics, the equivalence of programs should be the equality of their denotations.

- As a subcase, two programs having a common reduct should definitely be equivalent. However, this will in general not be complete: values may contain functions that are not identical, but perhaps reduce to the same value whenever applied to equivalent arguments.

- This leads to the notion of *observational equivalence.* Two expressions are observationally equivalent (at some observable type, such as integers) if their are indistinguishable whenever they are put in arbitrary (well-typed) contexts of the observable type.

End of parenthesis.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{D-OVar} \qquad \frac{\Gamma \vdash q : \forall \alpha.\, \sigma}{\Gamma \vdash q\ \tau : [\alpha \mapsto \tau]\sigma}\ \textsc{D-Inst} \qquad \frac{\Gamma \vdash q_1 : Q_1 \Rightarrow Q_2 \qquad \Gamma \vdash q_2 : Q_1}{\Gamma \vdash q_1\ q_2 : Q_2}\ \textsc{D-App}$$

Figure 7.3: Typing rules for dictionaries

For instance, two different elaborations algorithms that consistently change the representation of dictionaries (*e.g.* by ordering records in reverse order), may be equivalent if we cannot observe the representation of dictionaries.

Returning to the coherence problem, the only source of non-determinism in Mini Haskell is the elaboration of dictionaries. Hence, to ensure coherence, it suffices that two dictionary *values* of the same type are always equal. This does not mean that there is a unique way of building dictionaries, but that all ways are equivalent as they eventually return the same dictionary.

7.2.6 Elaboration of dictionaries

The elaboration of dictionaries is based on typing rules of System F—but restricted to a subset of the language. The relevant typing rules are given in Figure 7.3. However, elaboration significantly differs from type inference since the judgment $\Gamma \vdash q : Q$ is used for inferring q rather than τ. The judgment can be read as: in type environment Γ, a dictionary of type Q can be constructed by the dictionary expression q. As for type inference, elaboration of dictionaries is simplified by finding an appropriate syntax-directed presentation of the typing rules—but directed by the structure of the type of the expected dictionary instead of expressions.

Elaboration is also driven by the bindings available in the typing environment. These may be dictionary constructors z^h, given by instance definitions; dictionary accessors u^{K}, given by class declarations; dictionary arguments z, given by the local typing context. This suggests the presentation of the typing rules in Figure 7.4.

Dictionary values Let us first consider the elaboration of dictionary *values*, *i.e.* dictionary expressions that do not use dictionary parameters or projections. Thus, their derivation may only use D-OVar-Inst. They are typed in the environment $\Gamma_{\vec{H}\vec{h}}$, which does not contain free *type* variables. They actually do not access dictionaries, and only use the environment $\Gamma_{\vec{h}}$. Hence, all occurrences of D-OVar-Inst are of the form:

$$\frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma_{\vec{h}} \qquad \Gamma_{\vec{h}} \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma_{\vec{h}} \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}\ \textsc{D-OVar-Inst}$$

D-OVar-Inst
$$\frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \dots\, P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma \qquad \forall i \in 1..n,\ \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}$$

D-Proj
$$\frac{u : \forall \alpha.\, \mathsf{K}'\ \alpha \Rightarrow \mathsf{K}\ \alpha \in \Gamma \qquad \Gamma \vdash q : \mathsf{K}'\ \tau}{\Gamma \vdash u\ \tau\ q : \mathsf{K}\ \tau}$$

D-Var
$$\frac{z : \mathsf{K}\ \alpha \in \Gamma}{\Gamma \vdash z : \mathsf{K}\ \alpha}$$

Figure 7.4: Algorithmic typing rules for dictionaries

and the premise $\Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i$ is itself recursively built in the same way with this single rule. This rule can be read as a recursive definition, where Γ is constant, Q is the input type of the dictionary, and q is the output dictionary. This reading is deterministic if there is no choice in finding $z : \forall \vec{\beta}.\, P_1 \Rightarrow \dots\, P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$ in Γ. The binding z can only be a binding z^h introduced as the elaboration of some class instance h at type $\Gamma\vec{\beta}$, Hence, it suffices that *instance definitions never overlap* for z^h to be uniquely determined; if recursively each q_i is unique, then $z\ \vec{\tau}\ \vec{q}$ also is. Under this hypothesis, the elaboration is always unique and therefore coherent.

Definition 3 (Overlapping instances) *Two instances* $\mathsf{inst}\ \forall \vec{\beta}_1.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_1\ \vec{\beta}_1)\ \{r_1\}$ *and* $\mathsf{inst}\ \forall \vec{\beta}_2.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_2\ \vec{\beta}_2)\ \{r_2\}$ *of a class* K overlap *if the type schemes* $\forall \vec{\beta}_1.\, \mathsf{K}\ (\mathsf{G}_1\ \vec{\tau}_1)$ *and* $\forall \vec{\beta}_2.\, \mathsf{K}\ (\mathsf{G}_2\ \vec{\tau}_2)$ *have a common instance,* i.e. *in the current setting, if* G_1 *and* G_2 *are equal.*

Overlapping instances are an inherent source of incoherence, as it means that for some type Q (in the common instance), a dictionary of type Q may (possibly) be built using two different implementations.

Dictionary expressions Dictionary expressions may compute on dictionaries: they may extract sub-dictionaries or build new dictionaries from other dictionaries received as argument. Indeed, in overloaded code, the exact type is not fully known at compile type, hence dictionaries must be passed as arguments, from which superclass dictionaries may be extracted (actually must be extracted, as we forbade to pass a class and one of its super class dictionaries simultaneously).

Dictionaries are typically typed in the typing environment $\Gamma_{\vec{H}\vec{h}}, \Gamma^h$ where Γ^h binds the local typing context, *i.e.* assumptions $z : \mathsf{K}'\ \beta$ about dictionaries received as arguments. Hence, rules D-Proj and D-Var may now apply, *i.e.* the elaboration of expressions uses the three rules of 7.4. This can still be read as a backtracking proof search algorithm. The proof search always terminates, since premises always have strictly smaller Q than the conclusion when using the lexicographic ordering of the height of τ and then the reverse order of class inheritance: when no rule applies, the search fails; when rule D-Var applies, the search ends

with a successful derivation; when rule D-Proj applies, the premise is called with a smaller problem since the height is unchanged and $\mathsf{K}' \, \vec{\tau}$ with $\mathsf{K}' \prec \mathsf{K}$; when D-Ovar-Inst applies, the premises are called at type $\mathsf{K}_i \, \tau_j$ where τ_j is subtype of $\vec{\tau}$, hence of a strictly smaller height.

Non determinism However, non-overlapping of class instances is no more sufficient to prevent from non determinism. For instance, the introductory example of §7.2.1 defines two instances EqInt and OrdInt where the later contains an instance of the former. Hence, a dictionary of type EqInt may be obtained, either directly as EqInt, or indirectly as *Eq* OrdInt, by projecting the *Eq* sub-dictionary of class *Ord Int*. In fact, the latter choice could then be reduced at compile time and be equivalent to the first one.

One could force more determinism by fixing a strategy for elaboration. Restrict the use of rule D-Proj to cases where Q is P–when D-OVar-Inst does not apply. However, since the two elaborations paths are equivalent, the extra flexibility is harmless and may perhaps be useful freedom for the compiler.

Example of elaboration In our introductory example, the typing environment $\Gamma_{\vec{H}\vec{h}}$ is (we remind both the informal and formal names of variables):

$$\begin{array}{rcll}
\text{equal} & \triangleq & u_{\mathit{equal}} & : \; \forall \alpha.\, \mathit{Eq}\; \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}, \\
\text{EqInt} & \triangleq & z_{\mathit{Eq}}^{\mathit{Int}} & : \; \mathit{Eq}\; \mathsf{int} \\
\text{EqList} & \triangleq & z_{\mathit{Eq}}^{\mathit{List}} & : \; \forall \alpha.\, \mathit{Eq}\; \alpha \Rightarrow \mathit{Eq}\; (\mathsf{list}\; \alpha) \\
\\
\text{EqOrd} & \triangleq & u_{\mathit{Eq}}^{\mathit{Ord}} & : \; \forall \alpha.\, \mathit{Ord}\; \alpha \Rightarrow \mathit{Eq}\; \alpha \\
\text{lt} & \triangleq & u_{\mathit{lt}} & : \; \forall \alpha.\, \mathit{Ord}\; \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}
\end{array}$$

When elaborating the body of the *search* function, we have to infer a dictionary for EqOrd X OrdX in the local context X, OrdX : *Ord* X. Using formal notations, dictionaries are typed in the environment Γ equal to $\Gamma_0, \alpha, z : \mathit{Ord}\; \alpha$. and EqOrd is $u_{\mathit{Eq}}^{\mathit{Ord}}$. We have the following derivation:

$$\textsc{D-Proj}\;\dfrac{\overset{\textsc{D-OVar-Inst}}{\Gamma \vdash z : u_{\mathit{Eq}}^{\mathit{Ord}} : \mathit{Ord}\; \alpha \to \mathit{Eq}\; \alpha} \qquad \overset{\textsc{D-Var}}{\Gamma \vdash z : \mathit{Ord}\; \alpha}}{\Gamma \vdash u_{\mathit{Eq}}^{\mathit{Ord}}\; \alpha\; z : \mathit{Eq}\; \alpha}$$

7.3 Implicitly-typed terms

Our presentation of Mini Haskell is explicitly typed. Since we remain within an ML-like type system where type schemes are not first-class, we may leave some type information implicit. But how much? Class declarations define both the structure of dictionaries—a record type definition and its accessors—and the type scheme of overloaded symbols. Since, we inferring type schemes is out of the scope of ML-like type inference, class declarations

must remain explicit. Instance definitions are turned into recursive polymorphic definitions, which in ML require type scheme annotations. So they instance definitions also remain explicit. Fortunately, all remaining core language expressions, *i.e.* the body of instance definitions and the final program expression can be left implicit.

For instance, the example program in the introduction can be rewritten more concisely.

$$
\begin{array}{l}
\texttt{class}\ \mathit{Eq}\ (X)\ \ \{\ \mathrm{equal} : X \to X \to \mathit{Bool}\ \}\\
\texttt{inst}\ \mathit{Eq}\ (\mathit{Int})\ \{\ \mathrm{equal} = \mathit{primEqInt}\ \}\\
\texttt{inst}\ \mathit{Eq}\ (\mathit{Char})\ \{\ \mathrm{equal} = \mathit{primEqChar}\ \}\\
\texttt{inst}\ \Lambda(X)\ \mathit{Eq}\ (X) \Rightarrow \mathit{Eq}\ (\mathit{List}\ (X))\\
\quad \{\ \mathit{Eq} = \lambda(l_1)\ \lambda(l_2)\ \texttt{match}\ l_1,\ l_2\ \texttt{with}\\
\qquad\qquad \mid\ [],[]\ \to \mathit{true} \mid\ [], \_\ \mid\ [], \_ \to \mathit{false}\\
\qquad\qquad \mid\ h_1 :: t_1,\ h_2 :: t_2 \to \mathit{Eq}\ h_1\ h_2\ \&\&\ \mathit{Eq}\ t_1\ t_2\ \}\\
\texttt{class}\ \mathit{Eq}\ (X) \Rightarrow \mathit{Ord}\ (X)\ \{\ \mathrm{lt} : X \to X \to \mathit{Bool}\ \}\\
\texttt{inst}\ \mathit{Ord}\ (\mathit{Int})\ \{\ \mathrm{lt} = (<)\ \}\\
\texttt{let rec}\ \mathit{search}\ x\ l = \texttt{match}\ l\ \texttt{with}\ [] \to \mathit{false} \mid h{::}t \to \mathrm{equal}\ x\ h\ \|\ \mathit{search}\ x\ t\\
\texttt{let}\ b = \mathit{search}\ \mathit{Int}\ 1\ [1;\ 2;\ 3];;
\end{array}
$$

The missing type information can rebuilt by type inference.

Type inference To perform type inference in Mini Haskell, the idea is to see dictionary types $\mathsf{K}\ \tau$, which can only appear in type schemes and not in types, as a type constraint to mean *"there exists a dictionary of type* $\mathsf{K}\ \alpha$*"*. That is, we may read the type scheme $\forall \vec{\alpha}. \vec{P} \Rightarrow \tau$ as the constraint type scheme $\forall \vec{\alpha}[\vec{P}].\tau$ where $\vec{P}$ is seen as a type predicate, say a dictionary predicate. Therefore, we extend constraints with dictionary predicates:

$$C ::= \ldots \mid \mathsf{K}\ \tau$$

On ground types, a constraint $\mathsf{K}\ \mathbf{t}$ is satisfied if one can build a dictionary of type $\mathsf{K}\ \mathbf{t}$ in the initial environment $\Gamma_{\vec{H}\vec{h}}$ (that contains all class and instance declarations)—formally, if there exists a dictionary expression q such that $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ \mathbf{t}$. Then satisfiability of class-membership constraints is (with its unfolded version on the right):

$$\textsc{Instance}\ \frac{\mathsf{K}\ \phi\tau}{\phi \vdash \mathsf{K}\ \tau} \qquad\qquad \textsc{Instance}\ \frac{\Gamma_{\vec{H}\vec{h}} \vdash \rho : \mathsf{K}\ \phi\tau}{\phi \vdash \mathsf{K}\ \tau}$$

We use entailment to reason with class-membership constraints. For every class declaration $\mathsf{class}\ \mathsf{K}_1\ \alpha_1, \ldots\ \mathsf{K}_n\ \alpha_n \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$, we have:

$$\mathsf{K}\ \alpha \Vdash \mathsf{K}_1\ \alpha_1 \wedge \ldots\ \mathsf{K}_n\ \alpha_n \qquad \textbf{(K1)}$$

This rule allows to decompose any set of simple constraints into a canonical one.

Proof: Assume $\phi \vdash \mathsf{K}\ \alpha$, *i.e.* by Rule Instance $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\phi\alpha)$ for some dictionary q. From the class declaration in $\Gamma_{\vec{H}\vec{h}}$, we know that $\mathsf{K}\ \alpha$ is a record type definition that contains

fields $u^{\mathsf{K}}_{\mathsf{K}_i}$ of type $\mathsf{K}_i\ \alpha_i$. Hence, the dictionary value q contains field values of types $\mathsf{K}_i\ (\phi\alpha)$. Therefore, we have $\phi \vdash \mathsf{K}_i\ \alpha$ for all i in $1..n$, which implies $\phi \vdash \mathsf{K}_1\ \alpha \wedge \ldots \mathsf{K}_n\ \alpha$.

For every instance definition $\mathsf{inst}\ \forall\vec{\beta}.\ \mathsf{K}_1\ \beta_1, \ldots \mathsf{K}_p\ \beta_p \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$, we have

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \equiv \mathsf{K}_1\ \beta_1 \wedge \ldots \mathsf{K}_p\ \beta_p \qquad \textbf{(K2)}$$

This rule allows to decompose any class constraint into a conjunction of simple constraints (*i.e.* of the form $\mathsf{K}\ \alpha$).

<u>Proof</u>: Let h be the above instance definition. We proof both directions separately:

Case $\dashv\Vdash$: Assume $\phi \vdash \mathsf{K}_i\ \beta_i$ for i in $\{1, \ldots p\}$. By Rule Instance, for each i, there exists a dictionary q_i such that $\Gamma_{\vec{H}\vec{h}} \vdash q_i : \mathsf{K}_i\ (\phi\beta_i)$. Hence, $\Gamma_{\vec{H}\vec{h}} \vdash x_h\ \vec{\beta}\ q_1\ \ldots\ q_p : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$, *i.e.* by Rule Instance $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$.

Case $\Vdash$: Assume, $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$. *i.e.* there exists a dictionary q such that $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$. By inversion of typing (and non-overlapping of instance declarations), the only way to build such a dictionary is by an application of z_h. Hence, q must be of the form $x_h\ \vec{\beta}\ q_1\ \ldots\ q_p$ with $\Gamma_{\vec{H}\vec{h}} \vdash q_i : \mathsf{K}_i\ (\phi\beta_i)$. By Rule Instance, this means $\phi \vdash \mathsf{K}_i\ \beta_i$ for every i, which implies $\phi \vdash \mathsf{K}_1\ \beta_1 \wedge \ldots \mathsf{K}_p\ \beta_p$.

Notice that the equivalence (K2) still holds in an open-world assumption where new instance clauses may be added later, because another future instance definition cannot overlap with existing ones.

If class instances may overlap, the $\Vdash$ direction does not hold anymore; the rewriting rule:

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \longrightarrow \mathsf{K}_1\ \beta_1 \wedge \ldots \mathsf{K}_p\ \beta_p$$

remains sound (the inverse entailment holds, and thus type inference still infer sound typings), but it is incomplete (type inference could miss some typings).

We also use the following equivalence: for every class K and type constructor G for which there is no instance of K:

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \equiv \mathsf{false} \qquad \textbf{(K3)}$$

This rule allows to report failure as soon as a constraint of the form $\mathsf{K}\ (\mathsf{G}\ \vec{\tau})$ for which there is not instance of K for G appears.

<u>Proof</u>: The $\dashv\Vdash$ direction is a tautology, so it suffices to prove the $\Vdash$ direction. By contradiction. Assume $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$. This implies the existence of a dictionary q such that $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$. Then, there must be some x_h in Γ whose type scheme is of the form $\forall\vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, *i.e.* there must be an instance of class K for G.

Notice that the equivalence is only an inverse entailment in an open world assumption: when there is not instance of K at type G, the rewriting rule $\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \longrightarrow \mathsf{false}$ remains sound, but it is incomplete.

We are now fully equipped for type inference. Constraint generation is unchanged: see Figure 5.6. A constraint type scheme can then always be decomposed into one of the form $\forall \bar{\alpha}[P_1 \wedge P_2].\, \tau$ where $\mathsf{ftv}(P_1) \in \bar{\alpha}$ and $\mathsf{ftv}(P_2) \mathrel{\#} \bar{\alpha}$. The constraints P_2 can then be extruded to the enclosing context if any, so that we are just left with P_1, and thus a well-formed type scheme $\forall \vec{\alpha}.\, \vec{P} \Rightarrow \tau$ with a typing context $\vec{P}$.

To check well-typedness of a program $\vec{H}\ \vec{h}\ a$, we must check that: each expression a^h and the expression a are well-typed, in the environment used to elaborate them. This amounts to checking:

- $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash a^h : \tau^h$ where τ^h is given. That is, that $\mathsf{def}\ \Gamma_{\vec{H}\vec{h}}, \Gamma^h\ \mathsf{in}\ (\!|\, a^h \,|\!) \leq \tau^h \equiv \mathsf{true}$ holds;
- $\Gamma_{\vec{H}\vec{h}} \vdash a : \tau$ for some τ. That is, that $\mathsf{def}\ \Gamma_{\vec{H}\vec{h}}\ \mathsf{in}\ \exists \alpha.\, (\!|\, a \,|\!) \leq \alpha \equiv \mathsf{true}$ holds.

However, typechecking is not sufficient: type reconstruction should also return an explicitly-typed term M than can in turn be elaborated into some term N of System F, *i.e.* such that $\Gamma \vdash a \rightsquigarrow M : \tau$.

Type reconstruction Type reconstruction can be performed as described in §5.3.4 by keeping persistent constraints during resolution. As in ML, there may be several ways to reconstruct programs, which we may solve by requesting explicitly-typed terms to be canonical and principal.

Coherence When the source language is implicitly-typed, the elaboration from the source language into System F code is the composition of type reconstruction with elaboration of explicitly typed terms.

Hence, even though the elaboration is coherent for explicitly-typed terms, this may not be true for implicitly-typed terms. There are two potential problems:

- The language has principal constrained type schemes, but the elaboration requests unconstrained type schemes.
- Ambiguities could be hidden (and missed) by non principal type reconstructions.

Toplevel unresolved constraints The restrictions we put on class declarations and instance definitions ensure that the type system has principal constrained schemes (and principal typing reconstructions).

However, this does not imply that there are principal *unconstrained* type schemes. For example, assume that the principal constrained type scheme is $\forall \alpha[\mathsf{K}\ \alpha].\, \alpha \to \alpha$ and the typing environment contains two instances of $\mathsf{K}\ \mathsf{G}_1$ and $\mathsf{K}\ \mathsf{G}_2$ of class K. Constraint-free

instances of this type scheme are $\mathsf{G}_1 \to \mathsf{G}_1$ and $\mathsf{G}_2 \to \mathsf{G}_2$ but $\forall \alpha.\, \alpha \to \alpha$ is certainly not one. Not only neither choice is principal, but worse, the two choices would elaborate in expressions with different (and non-equivalent) semantics. Elaboration should fail in such cases.

This problem may appear while typechecking the final expression a in $\Gamma_{\vec{H}\vec{h}}$ that request an unconstrained type scheme $\forall \alpha.\, \tau$ It may also occur when typechecking the body of an instance definition h, which requests an explicit type scheme $\forall \vec{\beta}[\vec{Q}].\, \tau$ in $\Gamma_{\vec{H}\vec{h}}$ or, equivalently, a type τ in $\Gamma_{\vec{H}\vec{h}}, \vec{\beta}, \vec{Q}$. Consider, for example:

$\texttt{class}\ \mathit{Num}\ (X)\ \{\ 0 : X,\ (+) : X \to X \to X\ \}$
$\texttt{inst}\ \mathit{Num}\ \mathit{Int}\ \{\ 0 = \mathit{Int}.(0),\ (+) = \mathit{Int}.(+)\ \}$
$\texttt{inst}\ \mathit{Num}\ \mathit{Float}\ \{\ 0 = \mathit{Float}.(0),\ (+) = \mathit{Float}.(+)\ \}$
$\texttt{let}\ \mathit{zero} = 0 + 0;$

The type of zero or $\mathit{zero} + \mathit{zero}$ is $\forall \alpha[\mathit{Num}\ \alpha].\, \alpha$ while several class instances are possible for $\mathit{Num}\ X$. The semantics of the program is thus undetermined. Another example is:

$\texttt{class}\ \mathit{Readable}\ (X)\ \{\ \mathrm{read} : \mathit{descr} \to X\ \}$
$\texttt{inst}\ \mathit{Readable}\ (\mathit{Int})\ \{\ \mathrm{read} = \mathit{read\_int}\ \}$
$\texttt{inst}\ \mathit{Readable}\ (\mathit{Char})\ \{\ \mathrm{read} = \mathit{read\_char}\ \}$
$\texttt{let}\ v = \mathrm{read}\ (\mathit{open\_in}())$

The type of v is $\forall \alpha[\mathit{Readable}\ \alpha].\, \mathsf{unit} \to \alpha$—and several classes are possible for $\mathit{Readable}\ \alpha$. This program is also rejected.

Inaccessible constraint variables In the previous examples, the incoherence arise from the obligation to infer unconstrained toplevel type schemes. A similar problem may occur with *isolated* constraints in a type scheme. For instance, assume that $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ elaborates to $\mathsf{let}\ x : \forall \alpha[\mathsf{K}\ \alpha].\, \mathsf{int} \to \mathsf{int} = N_1\ \mathsf{in}\ N_2$. All applications of x in N_2 will lead to an unresolved constraint $\mathsf{K}\ \alpha$ for some fresh α since neither the argument nor the context of this application can determine the value of the type parameter α. Still, a dictionary of type $\mathsf{K}\ \tau$ must be given before N_1 can be executed.

Although x may not be used in N_2, in which case, all elaborations of the expression may be coherent, we may still raise an error, since an unusable local definition is certainly useless, hence probably a programmer's mistake. The error may then be raised immediately, at the definition site, instead of at every use of x.

The open-world view When there is a single instance $\mathsf{K}\ \mathsf{G}$ for a class K that appears in an unresolved or isolated constraint $\mathsf{K}\ \alpha$, the problem formally disappears, as all possible type reconstructions are coherent.

However, we may still not accept this situation, for modularity reasons, as an extension of the program with another non-overlapping *correct* instance declaration would make the program become ambiguous.

Formally, this amounts to saying that the program must be coherent in its current form, but also in all possible extensions with well-typed class definitions. This is taking an *open-world* view.

On the importance of principal type reconstruction A source of incoherence is when some class constraint remains undetermined. Some (usually arbitrary) less general elaboration could cover the problem—but the source program would remain incoherent. Hence, in order to detect programs with ambiguous semantics, it is essential that type reconstruction is principal. A program can still be specialized but only after it has been proved coherent. This freedom may actually be very useful for optimizations. Consider for example, the program

$$\texttt{let}\ \mathit{twice} = \lambda(x)\ x + x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

whose principal type reconstruction is:

$$\begin{array}{l}\texttt{let}\ \mathit{twice} : \forall(X)\ [\ \mathit{Num}\ X\ ]\ X \to X = \Lambda(X)\ [\mathit{Num}\ X]\ \lambda(x)\ x + x\ \texttt{in} \\ \mathit{twice}\ \mathit{Int}\ (\mathit{twice}\ \mathit{Int})\ 1\end{array}$$

This program is coherent. It's natural elaboration is

$$\begin{array}{l}\texttt{let}\ \mathit{twice}\ X\ \mathit{NumX} = \lambda(x : X)\ x\ (\mathit{plus}\ \mathit{NumX})\ x\ \texttt{in} \\ \mathit{twice}\ \mathit{Int}\ \mathrm{NumInt}\ (\mathit{twice}\ \mathit{Int}\ \mathrm{NumInt}\ 1)\end{array}$$

However, it can also be elaborated to

$$\texttt{let}\ \mathit{twice} = \lambda(x : \mathit{Int})\ x\ (\mathit{plus}\ \mathrm{NumInt})\ x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

avoiding the generalization of twice; moreover, the overloaded application $\mathit{plus}\ \mathrm{NumInt}$ can now be statically reduced, leading to:

$$\texttt{let}\ \mathit{twice} = \lambda(x : \mathit{Int})\ x\ \mathit{Int}.(+)\ x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

Overloading by return types All previous ambiguous examples are overloaded by their return types: For instance, in $0 : X$, the value 0 has an overloaded type that is not constraint by the argument; in $\mathrm{read} : \mathit{descr} \to X$, the return type is under specified, independently of the type of the argument.

To avoid such cases, Odersky et al. has suggested to prevent overloading by return types by requesting that overloaded symbols of a class $\mathsf{K}\ \alpha$ have types of the form $\alpha \to \tau$. The above examples would then be rejected by this definition.

In fact, disallowing overloading by return types—in addition to our previous restrictions—suffices to ensure that all well-typed programs are coherent. Moreover, untyped programs can then be given a direct semantics (which of course coincides with the semantics obtained by elaboration). Many interesting examples of overloading actually fits in this restricted subset. However, overloading by returns types is also found useful in several cases and Haskell allows it, using default rules to resolve ambiguities. This is still an arguable design choice in the Haskell community.

7.4 Variations

Changing the representation of dictionaries An overloaded method call u of a class K is elaborated into an application $u\ q$ of u to a dictionary expression q of class K. The function u and the representation of the dictionary are both defined in the elaboration of the class K and need not be known at the call site. This leaves some flexibility in the representation of dictionaries. For example, we have used records to represent dictionaries, but tuples would have been sufficient.

Going one step further, dictionaries need not contain the methods themselves but enough information from which the methods may be recovered. For example, dictionaries may be replaced by a derivation tree that proves the existence of the dictionary. This derivation tree may be concisely represented and passed around instead of the dictionary itself and be used and interpreted at at the call site to dispatch to the appropriate implementation of the method. Such an approach has been followed by Furuse (2003b).

This change of representation can also elegantly be explained as a type preserving compilation of dictionaries called concretization and described in Pottier and Gauthier (2006). It is somehow similar to defunctionalization and also requires that the target language is equipped with GADT (Guarded Abstract Data Types).

Multi-parameter type classes To allow multi-parameter type classes, we may extend the syntax of class definitions as follows:

$$\mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \vec{\alpha}\ \{\rho\}$$

where free variables of $\vec{P}$ must be bound in $\vec{\alpha}$. The current framework can easily be extended to handle multi-parameter type classes. For example, Collections may be represented by a type C whose elements are of type E and defined as follows:

$\texttt{class}\ \mathit{Collection}\ C\ E\ \{\ \mathrm{find} : C \to E \to \mathit{Option}(E),\ \mathrm{add} : C \to E \to C\ \}$
$\texttt{inst}\ \mathit{Collection}\ (\mathit{List}\ X)\ X\ \{\ \mathrm{find} = \mathit{List.find},\ \mathrm{add} = \lambda(c)\lambda(e)\ e{::}c\ \}$
$\texttt{inst}\ \mathit{Collection}\ (\mathit{Set}\ X)\ X\ \{\ \ldots\ \}$

However, the class *Collection* does not provide the intended intuition that collections are homogeneous. Indeed, we may define:

$\texttt{let}\ \mathit{add2}\ c\ x\ y = \mathrm{add}\ (\mathrm{add}\ c\ x)\ y$
$\mathit{add2} : \forall(C, E, E')\ \mathit{Collection}\ C\ E,\ \mathit{Collection}\ C\ E' \Rightarrow C \to E \to E' \to C$

This is accepted assuming that collections are heterogeneous. Although, this is unlikely the case, no contradiction can be assumed. However, if collections are indeed homogeneous, no instance of heterogeneous collections will ever be provided and the above code is overly general. As a result, uses of collections have unresolved often parameters, which would be resolved, if we had a way to tell the system that collections are homogeneous.

The solution is to add a clause to say that the parameter C determines the parameter E:

$$\texttt{class}\ \mathit{Collection}\ C\ E \mid C \to E\ \{\ \dots\ \}$$

Then, because C determines E, the two instances E and E' must be equal in C. Type dependencies also reduce overlapping between class declarations, since fewer instances of a class make sense. Hence they also allow example that would have to be rejected if type dependencies could not be expressed.

Associated types Associated types are an alternative to functional dependencies. They allow a class to declare its own *type* functions. Correspondingly, instance definitions must provide a definition for all associated types—in addition to values for overloaded symbols.

For example, the *Collection* class becomes a single parameter class with an associated type definition:

$$\begin{array}{l}
\texttt{class}\ \mathit{Collection}\ E\ \{ \\
\quad \texttt{type}\ C : * \to * \\
\quad \mathrm{find} : C \to E \to \mathit{Option}\ E \\
\quad \mathrm{add} : C \to E \to C \\
\} \\
\texttt{inst}\ \mathit{Collection}\ \mathit{Eq}\ X \Rightarrow \mathit{Collection}\ X\ \{\texttt{type}\ C = \mathit{List}\ E, \dots\ \} \\
\texttt{inst}\ \mathit{Collection}\ \mathit{Eq}\ X \Rightarrow \mathit{Collection}\ X\ \{\texttt{type}\ C = \mathit{Set}\ E, \dots\ \}
\end{array}$$

Associated types increase the expressiveness of type classes.

Overlapping instances In practice, overlapping instances may be desired! This seems in contradiction with the fact that overlapping instances are a source of incoherence. For example, one could provide a generic implementation of sets provided an ordering relation on elements, but also provide a more efficient version for bit sets. When overlapping instances are allowed, further rules are needed to disambiguate the overloading resolution and preserve coherence. For instance, priority rules may be used. An interesting resolution strategy is to give priority to the most specific match.

However, the semantics depend on some particular resolution strategy and becomes more fragile. See Jones et al. (1997) for a discussion. See also Morris and Jones (2010) for a recent new proposal. For example, the definitions:

$$\begin{array}{l}
\texttt{inst}\ \mathit{Eq}(X)\ \{\ \mathrm{equal} = (=)\ \} \\
\texttt{inst}\ \mathit{Eq}(\mathit{Int})\ \{\ \mathrm{equal} = \mathit{primEqInt}\ \}
\end{array}$$

could elaborate into the creation of both a generic dictionary and a specialized one.

$$\begin{array}{l}
\texttt{let}\ \mathit{Eq}\ X : \mathit{Eq}\ X = \{\ \mathrm{equal} = (=)\ \} \\
\texttt{let}\ \mathrm{EqInt} : \mathit{Eq}\ \mathit{Int} = \{\ \mathrm{equal} = \mathit{primEqInt}\ \}
\end{array}$$

Then, EqInt or *Eq Int* are two dictionaries of type *Eq Int* but with different implementations.

Restriction that are harder to lift We have made several restrictions to the definition of type classes. Some can be lifted at the price of some tolerable complication. Relaxing other restrictions, even if it could make sense in theory, would raise serious difficulties in practice.

For example, allowing constrained type schemes of the form $\mathsf{K}\ \tau$ instead of the restricted form $\mathsf{K}\ \alpha$ would affect many aspects of the language and it would becomes much more difficult to control the termination of constrained resolution and of the elaboration of dictionaries.

Allowing class instances of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ \tau\ \{\rho\}$ where τ is $\mathsf{G}\ \vec{\tau}$ and not just $\mathsf{G}\ \vec{\beta}$, it would become difficult to check non-overlapping of class instances.

Implicit values

Implicit values are a mecanism that allows to build values from types. The implies a way to populate an environement of definitions that can be used to build implicit values and a mecanism to introduce place holders where values should be build from their types.

Implicits values have been used in the language Scala for implicit conversions Sca (but they can do more). An extension of OCaml with implicit values is beeing prototyped. Implicit values have also been proposed as an alternative to Haskell type classes Oliveira et al. (2012).

Conclusions

Methods as overloading functions One approach to object-orientation is to see methods as overloaded functions. Then, objects carry class tags that can be used at runtime to find the best matching definition. This approach has been studied in detail by Millstein and Chambers (1999). See also Bonniot (2002, 2005).

Summary Static overloading is not a solution for polymorphic languages. Dynamics overloading must be used instead. The implementation of type classes in the Haskell language has proved quite effective: it is a practical, general, and powerful solution to dynamic overloading. Moreover, it works relatively well in combination with ML-like type inference.

However, besides the simplest case of overloading on which every one agrees, some useful extensions often come with serious drawbacks, and they is not yet an agreement on the best design compromises. In Haskell, the design decisions have often been in favor of expressiveness, but then loosing some of the properties and the canonicity of the minimalistic initial design.

Dynamic overloading is a typical and very elegant use of elaboration. The programmer could in principle write the elaborated program manually, explicitly building and passing

dictionaries around, but this would be cumbersome, tricky, error prone, and it would significantly obfuscate the code. Instead, the elaboration mechanism does this automatically, without arbitrary choices (in the minimal design) and with only local transformations that preserve the structure of the source program.

Further reading For an all-in-one explanation of Haskell-like overloading, see *The essence of Haskell* by Odersky et al. See also the Jones's monograph *Qualified types: theory and practice.* For a calculus of overloading see the ML& calculus proposed by Castagna (1997).

Recently, type classes have also been added to Coq Sozeau and Oury (2008). Interestingly, the elaboration of proof terms need not be coherent which makes it a simpler situation for overloading.

7.5 Ommitted proofs and answers to exercises

Solution of Exercise 42

We first need to show that the $\delta_\exists$ preserves typings. Assume that

$$\Gamma \vdash \mathsf{unpack}_{\exists\alpha.\tau_1} \, (\mathsf{pack}_{\exists\alpha.\tau} \, \tau' \, V) : \tau_0$$

By inversion of typing, τ_1 and τ_0 must be equal to τ and $\forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$, respectively, and the judgment $\Gamma \vdash V : [\alpha \mapsto \tau']\tau$ must hold. Let Γ' be $\Gamma, \beta, y : \forall\alpha.\,\tau \to \beta$. By weakening, we have $\Gamma' \vdash V : [\alpha \mapsto \tau']\tau$. We then have $\Gamma' \vdash y\,\tau'\,V : \beta$ and finally, we have

$$\Gamma \vdash \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\,\tau'\,V : \tau_0$$

as expected.

We then need to show that $\delta_\exists$ satisfies progress, *i.e.*, a full well typed application of $\mathsf{unpack}_{\exists\alpha.\tau}$ can always be reduced. Assume that $\Gamma \vdash \mathsf{unpack}_{\exists\alpha.\tau}\,V : \tau_0$. By inversion of typing, we must have $\Gamma \vdash V : \exists\alpha.\,\tau$. By the classification lemma (to be extended and rechecked), V must be an existential value, *i.e.* of the form $\mathsf{pack}_{\exists\alpha.\tau_1}\,\tau_0\,V_0$. Hence, $\mathsf{unpack}_{\exists\alpha.\tau}\,V$ reduces by $\delta_\exists$. ■

Solution of Exercise 43

We just force τ_1 to coincide with τ:

$$\mathsf{unpack}_{\exists\alpha.\tau}\,(\mathsf{pack}_{\exists\alpha.\tau}\,\tau'\,V) \;\longrightarrow\; \Lambda\beta.\lambda y{:}\forall\alpha.\,\tau \to \beta.\,y\,\tau'\,V \qquad (\delta_\exists)$$

The proof of subject reduction will know by construction that τ_0 is τ instead of learning it by inversion of typing. Conversely for progress, we will have to show that τ_1 and τ are equal by inversion so that $\delta_\exists$ can be applied. ■

Solution of Exercise 45

Let M_1 be $\mathsf{if}\ M\ \mathsf{then}\ V_1\ \mathsf{else}\ V_2$ where V_i is of the form $\mathsf{pack}\ \tau_i, V_i\ \mathsf{as}\ \exists\alpha\tau$ and the two witnesses τ_1 and τ_2 differ. There is no common type for the unpacking of the two possible results V_1 and V_2. The choice between those two possible results must be made, by evaluating M_1, before unpacking. ■

Solution of Exercise 47

The answer is in the 2007–2008 exam. ■

Chapter 8

Logical Relations

8.1 Introduction

Logical relations are relations between well-typed programs defined inductively on the structure of types.

There are two kinds of logical relations: unary and binary. Unary relations are predicate on expressions, while binary relations relates two expressions of the same type.

What they can be used for: unary relations can be used to prove type safety and strong normalisation; binary relations can be used to prove equivalence of programs and non-interferance properties.

Logical relations are a common proof method for programming language researchers that every one ought to know.

8.2 Normalization of simply-typed λ-calculus

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 4 *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 37 *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 38 *If $\varnothing \vdash M : \tau$ and $M \longrightarrow M'$, then $M \in \mathcal{T}_\tau$ iff $M' \in \mathcal{T}_\tau$.*

(Proof p. 173)

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

Lemma 39 *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 40 (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:

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

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.

8.3 Proofs and Solution to Exercises

Proof of Lemma 38

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.

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.

▷ Ken-etsu Fujita and Aleksy Schubert. Existential type systems with no types in terms. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 112–126, 2009. doi: 10.1007/978-3-642-02273-9_10.

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* $1, 2, \ldots, \omega$. 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. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

▷ 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.

Morten Heine Sørensen and Pawel Urzyczyn. *Studies in Logic and the Foundations of Mathematics*, chapter Lectures on the Curry-Howard Isomorphism. Elselvir Science Inc, 2006.

▷ 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.