Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of October 20, 2020
## Contents

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

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

3 **Simply-typed lambda-calculus** ........................................... 21
   3.1 Syntax .............................................................................. 21
   3.2 Dynamic semantics ........................................................... 21
   3.3 Type system ....................................................................... 22
   3.4 Type soundness .................................................................. 25
      3.4.1 Proof of subject reduction ............................................. 26
      3.4.2 Proof of progress ......................................................... 28
   3.5 Simple extensions ............................................................. 30
      3.5.1 Unit .......................................................................... 30
      3.5.2 Boolean ...................................................................... 30
      3.5.3 Pairs .......................................................................... 31
      3.5.4 Sums ......................................................................... 32
      3.5.5 Modularity of extensions .............................................. 32
      3.5.6 Recursive functions .................................................... 33
      3.5.7 A derived construct: let-bindings ................................. 33
   3.6 Exceptions ......................................................................... 35
      3.6.1 Semantics ................................................................... 35
CONTENTS

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

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

7 Overloading ............................................................ 145
7.1 An overview ................................................................. 145
7.1.1 Why use overloading? .................................................. 145
7.1.2 Different forms of overloading ...................................... 146
7.1.3 Static overloading ...................................................... 147
7.1.4 Dynamic resolution with a type passing semantics .......... 147
7.1.5 Dynamic overloading with a type erasing semantics ........ 148
7.2 Mini Haskell ................................................................. 149
7.2.1 Examples in \( \text{MH} \) .................................................... 149
7.2.2 The definition of Mini Haskell ..................................... 150
7.2.3 Semantics of Mini Haskell ......................................... 152
7.2.4 Elaboration of expressions ......................................... 154
7.2.5 Summary of the elaboration ........................................... 155
7.2.6 Elaboration of dictionaries ........................................... 157
7.3 Implicitly-typed terms .................................................. 159
7.4 Variations ................................................................. 165
7.5 Ommitted proofs and answers to exercises ....................... 169

8 Logical Relations .......................................................... 171
8.1 Introduction ............................................................... 171
8.2 Normalization of simply-typed λ-calculus ......................... 171
8.3 Proofs and Solution to Exercises .................................... 173
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 $\lambda$-calculus, ML, and System $\mathcal{F}$, of increasing expressiveness. In each case, we have presented an explicitly-typed and an implicitly-typed version of the language and shown a close correspondence between the two views, thanks to a type-erasing semantics.

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

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

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

This way, types need never be guessed, just looked up into the typing context, instanti-
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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-calculus

The type inference algorithm for simply-typed \( \lambda \)-calculus, is due to Hindley. The idea behind the algorithm is simple. Because simply-typed \( \lambda \)-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 \( \lambda \)-calculus decomposes into constraint generation followed by constraint solving. Simple types are first-order terms. Thus, solving a collection of equations between simple types is first-order unification. First-order unification can be performed incrementally in quasi-linear time, and admits particularly simple solved forms.
5.2. TYPE INFERENCE FOR SIMPLY-TYPED \( \lambda \)-CALCULUS

\[
\begin{align*}
\langle \Gamma \vdash x : \tau \rangle & = \Gamma(x) = \tau \\
\langle \Gamma \vdash \lambda x. a : \tau \rangle & = \exists \alpha_1 \alpha_2. (\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle \land \tau = \alpha_1 \to \alpha_2) \quad \text{if } \alpha_1, \alpha_2 \not\in \Gamma, \tau \\
\langle \Gamma \vdash a_1 a_2 : \tau \rangle & = \exists \alpha. (\langle \Gamma \vdash a_1 : \alpha \to \tau \rangle \land \langle \Gamma \vdash a_2 : \alpha \rangle) \quad \text{if } \alpha \not\in \Gamma, \tau
\end{align*}
\]

Figure 5.1: constraint generation for simply-typed \( \lambda \)-calculus

5.2.1 Constraints

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

There are two syntactic categories: types and constraints.

\[
\begin{align*}
\tau & ::= \alpha \mid F \bar{t} \\
C & ::= \text{true} \mid \text{false} \mid \tau = \tau \mid C \land C \mid \exists \alpha. C
\end{align*}
\]

A type is either a type variable \( \alpha \) 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:

\[
t ::= F \bar{t}
\]

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 \( \phi \) is a total mapping of type variables to ground types. By homomorphism, a ground assignment determines a total mapping of types to ground types.

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

\[
\begin{align*}
\phi \vdash \text{true} & \quad \phi \vdash \tau_1 = \tau_2 & \quad \phi \vdash C_1 \quad \phi \vdash C_2 & \quad \phi \vdash \exists \alpha. C \\
\phi \vdash \tau_1 = \tau_2 & \quad \phi \vdash C_1 \land C_2 & \quad \phi \vdash [\alpha \mapsto t] \vdash C
\end{align*}
\]

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

Type inference is reduced to constraint solving by defining a mapping \( \langle \Gamma \vdash a : \tau \rangle \) of candidate judgments to constraints, as given in Figure 5.11. Thanks to the use of existential quantification, the names that occur free in \( \langle \Gamma \vdash a : \tau \rangle \) are a subset of those that occur free in \( \Gamma \) or \( \tau \). This allows the freshness side conditions to remain local—there is no need to informally require “globally fresh” type variables.
5.2.2 A detailed example

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

Solving the constraint means to find all possible ground assignments for \( \alpha_0 \) that satisfy the constraint. Typically, this is done by transforming the constraint into successive equivalent constraints until some constraint that is obviously satisfiable and from which solutions may be directly read.

Performing constraint generation for the 3 \( \lambda \)-abstractions, we have:

\[
C' = \exists \alpha_1 \alpha_2 . \left( \exists \alpha_3 \alpha_4 , \left( \exists \alpha_5 \alpha_6 . \left( \langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash ( f x , f y ) : \alpha_6 \rangle \right) \right) \right)
\]

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

\[
( \exists \alpha . C_1 ) \land C_2 \equiv \exists \alpha . ( C_1 \land C_2 ) \quad \text{if } \alpha \neq C_2
\]

Hence, hoisting \( \alpha_3 \) and \( \alpha_4 \), and \( \alpha_5 \) and \( \alpha_6 \) twice, we get:

\[
C' \equiv \exists \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 . \left( \langle \Gamma \vdash ( f x , f y ) : \alpha_6 \rangle \left( \alpha_4 = \alpha_5 \rightarrow \alpha_6 \land \alpha_2 = \alpha_3 \rightarrow \alpha_4 \land \alpha_0 = \alpha_1 \rightarrow \alpha_2 \right) \right)
\]

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

\[
\exists \alpha . ( C \land \alpha = \tau ) \equiv [\alpha \mapsto \tau ] C
\]

if \( \alpha \neq \tau \)

By successive elimination of \( \alpha_4 \) then \( \alpha_2 \), we get:

\[
C' \equiv \exists \alpha_1 \alpha_2 \alpha_3 \alpha_5 \alpha_6 . \left( \langle \Gamma \vdash ( f x , f y ) : \alpha_6 \rangle \right)
\]

Let us now perform constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable \( ( \alpha_6 ) \).

\[
C' \equiv \exists \left\{ \alpha_1 \alpha_2 \alpha_3 \alpha_5 \alpha_6 \alpha_7 \alpha_8 \right\} . \left( \langle \Gamma \vdash f x : \alpha_7 \rangle \right) \left( \langle \Gamma \vdash f y : \alpha_8 \rangle \right) \left( \alpha_7 \times \alpha_8 = \alpha_6 \right) \left( \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_6 = \alpha_0 \right) \equiv \exists \left\{ \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8 \right\} . \left( \langle \Gamma \vdash f x : \alpha_7 \rangle \right) \left( \langle \Gamma \vdash f y : \alpha_8 \rangle \right) \left( \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8 = \alpha_0 \right)
\]
Let us focus on the first application, perform constraint generation for the variables \( f \) and \( x \) (recall that \( \Gamma \) stands for \((f : \alpha_1; x : \alpha_3; y : \alpha_5)\)), and eliminate a type variable \((\alpha_9)\):

\[
C_1 = \exists \alpha_9. \left( \frac{\Gamma \vdash f : \alpha_9 \rightarrow \alpha_7}{\Gamma \vdash x : \alpha_9} \right) = \exists \alpha_9. \left( \frac{\alpha_1 = \alpha_9} {\alpha_3 = \alpha_9} \right) \equiv \alpha_1 = \alpha_3 \rightarrow \alpha_7 = C_2
\]

Applying this simplification under a context, with the rule:

\[
C_1 \equiv C_2 \Rightarrow \mathcal{R}[C_1] \equiv \mathcal{R}[C_2]
\]

we have:

\[
C = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \left( \begin{array}{c}
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8
\end{array} \right)
\]

We may simplify the right-hand application analogously:

\[
C = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \left( \begin{array}{c}
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8
\end{array} \right)
\]

We may apply transitivity at \(\alpha_1\), structural decomposition, and eliminate three type variables \((\alpha_1, \alpha_5, \alpha_8)\):

\[
C = \exists \alpha_1 \alpha_3 \alpha_5 \alpha_7 \alpha_8. \left( \begin{array}{c}
\alpha_1 = \alpha_3 \rightarrow \alpha_7 \\
\alpha_0 = \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_7 \times \alpha_8
\end{array} \right)
\]

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

\[
\varnothing \vdash \lambda f x y. (f x, f y) : \alpha_0
\]

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

\[
(t_3 \rightarrow t_7) \rightarrow t_3 \rightarrow t_3 \rightarrow t_7 \times t_7
\]

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

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

```ocaml
# fun f x y -> (f x, f y);

- : ('a -> 'b) -> 'a -> 'a -> 'b * 'b = (fun)
```

This technique is used also by Standard ML and Haskell.

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

\[ \langle f : \alpha_1 \vdash \lambda xy. (f x, f y) : \alpha_2 \rangle \equiv \exists \alpha_3 \ldots (\alpha_1 = \alpha_3 \rightarrow \alpha_7 \land \alpha_2 = \alpha_3 \rightarrow \alpha_3 \rightarrow \alpha_7 \times \alpha_7) \]

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

\[(f : t_3 \rightarrow t_7), \ t_3 \rightarrow t_3 \rightarrow t_7 \times t_7)\]

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

### 5.2.3 Soundness and completeness of type inference

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

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

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

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

In other words, assuming \( \text{dom}(\Gamma) = \text{fv}(a) \), \( \phi \) satisfies the constraint \( \langle \Gamma \vdash a : \tau \rangle \) if and only if \( (\phi \Gamma, \phi \tau) \) is a (ground) typing of \( a \). The direct implication is soundness; the reverse implication is completeness. The proof is by structural induction over \( a \). (Proof p. 123)

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

**Corollary 30** Let \( \text{fv}(a) = \{x_1, \ldots, x_n\} \), where \( n \geq 0 \). Let \( \alpha_0, \ldots, \alpha_n \) be pairwise distinct type variables. Then, the ground typings of \( a \) are described by \( ((x_i : \phi_i)_{i \in 1..n}, \phi_0) \) where \( \phi \) ranges over all solutions of \( \langle (x_i : \alpha_i)_{i \in 1..n} \vdash a : \alpha_0 \rangle \).

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

### 5.2.4 Constraint solving

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

Our constraints are equations on first-order terms. They can be solved by first-order unification. The algorithm can be described as constraint solving. However, in order to
5.2. TYPE INFERENCE FOR SIMPLY-TYPED $\lambda$-CALCULUS

$$(\exists \bar{\alpha}.U_1) \wedge U_2 \rightarrow \exists \bar{\alpha}.(U_1 \wedge U_2)$$  
if $\bar{\alpha} \neq U_2$

$$(\exists \bar{\alpha}.U_1) \wedge \alpha = \epsilon \wedge \alpha = \epsilon' \rightarrow \alpha = \epsilon = \epsilon'$$  
(fusion)

$$F\bar{\alpha} = F\bar{\tau} = \epsilon \rightarrow \bar{\alpha} = \bar{\tau} \wedge F\bar{\alpha} = \epsilon$$  
(decomposition)

$$F\tau_1 \ldots \tau_i \ldots \tau_n = \epsilon \rightarrow \exists \alpha. (\alpha = \tau_i \wedge F\tau_1 \ldots \alpha \ldots \tau_n = \epsilon)$$  
(naming)

if $\tau_i$ is not a variable $\wedge \alpha \neq \tau_1, \ldots, \tau_n, \epsilon$

$$F\bar{\tau} = F'\bar{\tau}' = \epsilon \rightarrow \text{false}$$  
(clash)

if $F \neq F'$

U $\rightarrow$ false  
(occurs check)

if $U$ is cyclic

$U[\text{false}]$ $\rightarrow$ false  
(error propag.)

$\alpha = \alpha = \epsilon \rightarrow \alpha = \epsilon$  
(elim dupl.)

$F\bar{\tau}$ $\rightarrow$ true  
(elim triv.)

$U \wedge \text{true}$ $\rightarrow$ U  
(elim true)

Figure 5.2: Solving unification constraints

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

$$U ::= \text{true} | \text{false} | \epsilon | U \wedge U | \exists \bar{\alpha}.U$$

A multi-equation $\epsilon$ is a multi-set of types. Its interpretation is given by

$$\forall \tau \in \epsilon, \quad \phi \tau = t$$

That is, $\phi$ satisfies $\epsilon$ if and only if $\phi$ maps all members of $\epsilon$ to a single ground type.

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

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

A solved form is either false or $\exists \bar{\alpha}.U$, 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 \( \tau \), and uses two global variables, \( \mathcal{V} \) and \( \varphi \). Variable \( \mathcal{V} \) is an infinite fresh supply of type variables; \( \varphi \) is an idempotent substitution (of types for type variables), initially the identity. The fresh primitive is defined as:

\[
\text{fresh} = \text{do } \alpha \in \mathcal{V}; \text{ do } \mathcal{V} \leftarrow \mathcal{V} \setminus \{\alpha\}; \text{ return } \alpha
\]

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

\[ J(\Gamma \vdash x) = \text{let } \forall \alpha_1 \ldots \alpha_n. \tau = \Gamma(x) \]
\[ \text{do } \alpha'_1, \ldots, \alpha'_n = \text{fresh} \ldots \text{fresh} \]
\[ \text{return } [\alpha_i \mapsto \alpha'_i]_i (\tau) - \text{take a fresh instance} \]
\[ J(\Gamma \vdash \lambda x.a_1) = \text{do } \alpha = \text{fresh} \]
\[ \text{do } \tau_1 = J(\Gamma; x : \alpha \vdash a_1) \]
\[ \text{return } \alpha \rightarrow \tau_1 - \text{form an arrow type} \]
\[ J(\Gamma \vdash a_1 a_2) = \text{do } \tau_1 = J(\Gamma \vdash a_1) \]
\[ \text{do } \tau_2 = J(\Gamma \vdash a_2) \]
\[ \text{do } \alpha = \text{fresh} \]
\[ \text{do } \varphi \leftarrow \text{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi \]
\[ \text{return } \alpha - \text{solve } \tau_1 = \tau_2 \rightarrow \alpha \]
\[ J(\Gamma \vdash \text{let } x = a_1 \text{ in } a_2) = \text{do } \tau_1 = J(\Gamma \vdash a_1) \]
\[ \text{let } \sigma = \forall \setminus \text{ftv}(\varphi(\Gamma)), \varphi(\tau_1) - \text{generalize} \]
\[ \text{return } J(\Gamma; x : \sigma \vdash a_2) \]

(\forall \setminus \bar{\alpha}. \tau \text{ quantifies over all type variables other than } \bar{\alpha}.)

Figure 5.3: Type inference algorithm for ML

5.3.2 Constraint-based type inference for ML

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

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

\[ C ::\ldots | x = \tau | \text{def } x : \tau \text{ in } C \]

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

\[ \text{def } x : \tau \text{ in } C \equiv [x \mapsto \tau]C \]

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

\[ \frac{\psi x = \phi \tau}{\phi, \psi \vdash x = \tau} \quad \frac{\phi, \psi [x \mapsto \phi \tau] \vdash C}{\phi, \psi \vdash \text{def } x : \tau \text{ in } C} \]

(All other rules are modified to just transport \(\psi\).) Constraint generation becomes a mapping of an expression \(a\) and a type \(\tau\) to a constraint \(\langle a : \tau \rangle\). There is no longer a need for the
\[
\begin{align*}
\langle x : \tau \rangle & = x = \tau \\
\langle \lambda x . a : \tau \rangle & = \exists \alpha_1 \alpha_2 . (\text{def } x : \alpha_1 \text{ in } \langle a : \alpha_2 \rangle \land \alpha_1 \rightarrow \alpha_2 = \tau) \quad \text{if } \alpha_1, \alpha_2 \neq a, \tau \\
\langle a_1 a_2 : \tau \rangle & = \exists \alpha . (\langle a_1 : \alpha \rightarrow \tau \rangle \land \langle a_2 : \alpha \rangle) \quad \text{if } \alpha \neq a_1, a_2, \tau
\end{align*}
\]

Figure 5.4: Constraints with program variables

parameter \( \Gamma \). Constraint generation is defined in Figure 5.4.

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

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

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

**Enriching constraints** To permit polymorphism, we must extend the syntax of constraints so that a variable \( x \) denotes not just a ground type, but a *set of ground types*. However, these sets cannot be represented as type schemes \( \forall \bar{\alpha}. \tau \), 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:

\[
\begin{align*}
C & ::= \tau \mid \tau \lor C \mid \exists \alpha . C \mid x \leq \tau \mid \sigma \leq \tau \mid \text{def } x : \sigma \text{ in } C \\
\sigma & ::= \forall \bar{\alpha}[C]. \tau
\end{align*}
\]

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

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

\[
\text{def } x : \sigma \text{ in } C \equiv [x \mapsto \sigma]C \quad (\forall \bar{\alpha}[C]. \tau) \leq \tau' \equiv \exists \bar{\alpha}. (C \land \tau = \tau') \quad \text{if } \bar{\alpha} \neq \tau'
\]

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

The interpretation of constraints must be redefined since the environment \( \psi \) now maps program variables to sets of ground types. The environment \( \phi \) still maps type variables to ground types. Hence, a type variable \( \alpha \) still denotes a ground type. A variable \( x \) now denotes
5.3. TYPE INFERENCE FOR ML

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

\[
\begin{align*}
\phi \tau \in \psi x & \quad \phi \tau \in (\psi \upsilon) \sigma & \quad \phi, \psi \vdash x \leq \tau \\
\phi, \psi \vdash \sigma \leq \tau & \quad \phi, \psi \vdash \text{def } x : \sigma \text{ in } C
\end{align*}
\]

The interpretation of \(\forall \alpha [C], \tau\) under \(\phi\) and \(\psi\), written \(\left(\psi^x\right)(\forall \alpha [C], \tau)\) is the set of all \(\phi' \tau\), where \(\phi\) and \(\phi'\) coincide outside \(\alpha\) and where \(\phi'\) and \(\psi\) satisfy \(C\):

\[
\left(\psi^x\right)(\forall \alpha [C], \tau) \triangleq \{\phi' \tau \mid (\phi' \setminus \alpha = \phi \setminus \alpha) \land (\phi', \psi \vdash C)\}
\]

If \(C\) is empty, then \(\left(\psi^x\right)(\forall \alpha [C], \tau)\) is \(\{(\phi[\alpha \mapsto \bar{t}]) \tau\}\). If \(\bar{\alpha}\) and \(C\) are empty, then \(\left(\psi^x\right)\tau\) is \(\phi \tau\).

For instance, the interpretation of \(\forall \alpha [\exists \beta. \alpha = \beta \rightarrow \gamma]. \alpha \rightarrow \alpha\) under \(\phi\) and \(\psi\) is the set of all ground types of the form \((t \rightarrow \phi \gamma) \rightarrow (t \rightarrow \phi \gamma)\), where \(t\) ranges over ground types. This is also the interpretation of an unconstrained typed scheme, namely \(\forall \beta. (\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)\).

In fact, this is a general situation:

**Lemma 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 \(C\), \(\text{def } x : \sigma \text{ in } C\) is equivalent to \(C\) — whether or not the constraints appearing in \(\sigma\) are solvable. To enforce the constraints in \(\sigma\) to be solvable, we use a variant of the \(\text{def}\) construct:

\[
\text{let } x : \sigma \text{ in } C \triangleq \text{def } x : \sigma \text{ in } (\exists \alpha. x \leq \alpha \land C')
\]

Expanding the constraint type scheme \(\sigma\) of the form \(\forall \alpha [C], \tau\) and simplifying, an equivalent definition is:

\[
\text{let } x : \forall \alpha [C], \tau \text{ in } C' \triangleq (\exists \alpha. C \land \text{def } x : \forall \alpha [C], \tau) \text{ in } C''
\]

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

\[
\left(\psi^x\right)\sigma \not\equiv \emptyset \quad \phi, \psi \vdash (\psi^x)\sigma \vdash C
\]

\[
\phi, \psi \vdash \text{let } x : \sigma \text{ in } C
\]

Constraint generation for ML is defined in Figure 5.5. The abbreviation \((a)\) is a principal constrained type scheme for \(a\): its intended interpretation is the set of all ground types that \(a\) admits.
\[
\begin{align*}
\langle x : \tau \rangle &= x \leq \tau \\
\langle \lambda x. a : \tau \rangle &= \exists \alpha_1 \alpha_2. (\text{def } x : \alpha_1 \text{ in } \langle a : \alpha_2 \rangle \land \alpha_1 \rightarrow \alpha_2 = \tau) \\
&\quad \text{if } \alpha_1, \alpha_2 \not= a, \tau \\
\langle a_1 a_2 : \tau \rangle &= \exists \alpha. (\langle a_1 : \alpha \rightarrow \tau \rangle \land \langle a_2 : \alpha \rangle) \\
&\quad \text{if } \alpha \not= a_1, a_2, \tau \\
\langle \text{let } x = a_1 \text{ in } a_2 : \tau \rangle &= \text{let } x : \langle a_1 \rangle \text{ in } \langle a_2 : \tau \rangle \\
\langle a \rangle &= \forall \alpha [\langle a : \alpha \rangle], \alpha 
\end{align*}
\]

Figure 5.5: Constraint generation for ML

Lemma 34 (Constraint equivalences) The following equivalences hold:

\[
\begin{align*}
(1) \quad \exists \alpha. (\langle a : \alpha \rangle \land \alpha = \tau) &\equiv \langle a : \tau \rangle \quad \text{if } \alpha \not= \tau \\
(2) \quad \langle a \rangle \leq \tau &\equiv \langle a : \tau \rangle \\
(3) \quad [x \mapsto \langle a_1 \rangle] \langle a_2 : \tau \rangle &\equiv \langle [x \mapsto a_1]a_2 : \tau \rangle 
\end{align*}
\]

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

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

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

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

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

Lemma 36 The size of \( \langle a : \tau \rangle \) is linear in the sum of the sizes of \( a \) and \( \tau \).

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

Theorem 17 (Soundness and completeness) Assume \( \text{fv}(a) = \text{dom}(\Gamma) \). Then, \( \phi, \phi \Gamma \vdash \langle a : \tau \rangle \) if and only if \( \phi \Gamma \vdash a : \phi \tau \).
Key points  Notice that constraint generation has linear complexity; constraint generation
and constraint solving are separate. This makes constraints suitable for use in an efficient
and modular implementation. In particular, the constraint language remains small as the
programming language grows.

5.3.3 Constraint solving by example

For our running example, assume that the initial environment \( \Gamma_0 \) stands for
\( \text{assoc} : \forall \alpha \beta. \alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta \). That is, the
constraints considered next are implicitly wrapped within the context \( \text{def } \Gamma_0 \in [] \). Let \( a \) stand for the term:
\[
\lambda x.\lambda l_1.\lambda l_2. \text{let } \text{assocx} = \text{assoc } x \text{ in } (\text{assocx } l_1, \text{assocx } l_2)
\]

One may anticipate that \( \text{assocx} \) receives a polymorphic type scheme, which is instantiated
twice at different types. Let \( \Gamma \) stand for \( x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2 \). Then, the constraint \( \{ a : \alpha \} \) is,
after a few minor simplifications:
\[
\exists \alpha_0 \alpha_1 \alpha_2 \beta.
\left( \begin{array}{c}
\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\
def \Gamma \text{ in }
\left( \begin{array}{c}
\text{let } \text{assocx} : \forall \gamma_1 \left[ \exists \gamma_2. \left( \text{assoc } \gamma_2 \leq \gamma_1 \rightarrow \gamma_1 \right) \right] \text{ in }
\exists \beta_1 \beta_2. \left( \begin{array}{c}
\beta = \beta_1 \times \beta_2 \\
\forall i \in \{1, 2\}, \exists \gamma_2. \left( \text{assocx } \gamma_2 \leq \beta_i \land l_i \leq \gamma_2 \right) \end{array} \right)
\end{array} \right)
\end{array} \right)
\]

Constraint solving can be viewed as a rewriting process that exploits equivalence laws.
Because equivalence is, by construction, a congruence, rewriting is permitted within an arbitrary
context. For instance, environment access is allowed by the law
\[
\text{let } x : \sigma \text{ in } \text{R}[x \leq \tau] \equiv \text{let } x : \sigma \text{ in } \text{R}[\sigma \leq \tau]
\]

where \( \text{R} \) is a context that does not bind \( x \). Thus, within the context \( \text{def } \Gamma_0; \Gamma \in [] \), we have
the following equivalence:
\[
\text{assoc } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2 \equiv \exists \alpha \beta. (\alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_2 \rightarrow \gamma_1) \land \alpha_0 = \gamma_2
\]

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

\[
\exists \gamma_2. (\exists \alpha \beta. (\alpha \rightarrow \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_2 \rightarrow \gamma_1) \land \alpha_0 = \gamma_2) \\
\equiv \exists \gamma_2. (\exists \alpha \beta. (\alpha = \gamma_2 \land \text{list} (\alpha \times \beta) \rightarrow \beta = \gamma_1) \land \alpha_0 = \gamma_2) \\
\equiv \exists \gamma_2. (\exists \beta. (\text{list} (\gamma_2 \times \beta) \rightarrow \beta = \gamma_1) \land \alpha_0 = \gamma_2) \\
\equiv \exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1)
\]

Hence,
\[
\forall \gamma_1 [\exists \gamma_2. (\text{assoc } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2)]. \gamma_1 \\
\equiv \forall \gamma_1 [\exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1)]. \gamma_1 \\
\equiv \forall \gamma_1 [\exists \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1)]. \gamma_1 \\
\equiv \forall \beta. (\text{list} (\alpha_0 \times \beta) \rightarrow \beta
\]
We have used the rule:

\[ \forall \alpha \exists \beta. C \land \tau \equiv \forall \alpha \beta[C] \land \tau \quad \text{if } \beta \neq \tau \]

The initial constraint has now been simplified down to:

\[ \exists \alpha \beta_0 \beta_2 \beta. \begin{align*}
\alpha &= \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\
\text{def } \Gamma \text{ in } &\exists \beta \text{. list } (\alpha_0 \times \beta) \to \beta \text{ in } \\
\exists \beta_1 \beta_2. &\begin{cases} 
\beta = \beta_1 \times \beta_2 \\
\forall i \in \{1, 2\}, \exists \gamma_2. (\text{assocx} \leq \gamma_2 \rightarrow \beta_i \land l_i \leq \gamma_2)
\end{cases}
\end{align*} \]

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 \( \exists \gamma_2. (\text{assocx} \leq \gamma_2 \rightarrow \beta_i \land l_i \leq \gamma_2) \) where \( i \in \{1, 2\} \), is rewritten:

\[ \exists \gamma_2. (\exists \beta. (\text{list } (\alpha_0 \times \beta) \to \beta = \gamma_2 \rightarrow \beta_i) \land \alpha_i = \gamma_2) \equiv \exists \beta. (\exists \beta. (\text{list } (\alpha_0 \times \beta) = \alpha_i \rightarrow \beta = \beta_i)) \equiv \exists \beta. (\text{list } (\alpha_0 \times \beta) = \alpha_i) \equiv \text{list } (\alpha_0 \times \beta_i) = \alpha_i \]

The initial constraint has now been simplified down to:

\[ \exists \alpha_0 \alpha_1 \alpha_2 \beta. \begin{align*}
\alpha &= \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\
\text{def } \Gamma \text{ in let } &\text{assocx: } \forall \beta. \text{list } (\alpha_0 \times \beta) \to \beta \text{ in } \exists \beta_1 \beta_2. (\beta = \beta_1 \times \beta_2 \\
&\forall i \in \{1, 2\}, \text{list } (\alpha_0 \times \beta_i) = \alpha_i)
\end{align*} \]

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

\[ \exists \alpha_0 \alpha_1 \alpha_2 \beta. \begin{align*}
\alpha &= \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\
\exists \beta_1 \beta_2. &\begin{cases} 
\beta = \beta_1 \times \beta_2 \\
\forall i \in \{1, 2\}, \text{list } (\alpha_0 \times \beta_i) = \alpha_i
\end{cases}
\end{align*} \]

that is, by extrusion:

\[ \exists \alpha_0 \alpha_1 \alpha_2 \beta_1 \beta_2. \begin{align*}
\alpha &= \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\
\beta &= \beta_1 \times \beta_2 \\
\forall i \in \{1, 2\}, \text{list } (\alpha_0 \times \beta_i) = \alpha_i
\end{align*} \]

Finally, by eliminating a few auxiliary variables:

\[ \exists \alpha_0 \beta_1 \beta_2. (\alpha = \alpha_0 \rightarrow \text{list } (\alpha_0 \times \beta_1) \rightarrow \text{list } (\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2) \]

We have shown the following equivalence between constraints:

\[ \text{def } \Gamma_0 \text{ in } \ll a : \alpha \gg \equiv \exists \alpha_0 \beta_1 \beta_2. (\alpha = \alpha_0 \rightarrow \text{list } (\alpha_0 \times \beta_1) \rightarrow \text{list } (\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2) \]

That is, the principal type scheme of \( a \) relative to \( \Gamma_0 \) is

\[ \ll a \gg = \forall \alpha \ll \ll a : \alpha \gg \gg, \alpha \equiv \forall \alpha_0 \beta_1 \beta_2. \alpha_0 \rightarrow \text{list } (\alpha_0 \times \beta_1) \rightarrow \text{list } (\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2 \]
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 \( J \), to computing a principal type scheme before examining the right-hand side of a let construct.

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

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

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

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

```ml
let mairson =
  let f = fun x → (x, x) in
  (... n times ...)  
  let f = fun x → f (f x) in
  f (fun z → z)
```

This term can be placed in the context `let x = ... in ()` to ignore the time spent outputting 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 \( \langle a \rangle \) as an abbreviation for \( \forall \alpha [\langle a : \alpha \rangle] \), \( \alpha \). However, using the equiv-
alence between \( \llangle a : \tau \rrangle \) and \( \llangle a \rrangle \preceq \tau \), we may conversely use principal constraint type schemes in place of program constraints. This leads to an alternative presentation of constraint generation described in Figure 5.6. (Compare it with the previous definition in Figure 5.5).

### 5.3.4 Type reconstruction

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

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

For example the implicitly-typed term
\[
\text{let } x = \lambda y. y \text{ in } x
\]
may be explicitly typed as either one of
\[
\begin{align*}
\text{let } x : \text{int} \to \text{int} &= \lambda y : \text{int}. y \text{ in } x \\
\text{let } x : \forall \alpha. \alpha \to \alpha &= \Lambda x : \text{int}. x \text{ in } x
\end{align*}
\]

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

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

for all decompositions of \( M \) into \( C[\text{let } x : \sigma = M_1 \text{ in } M_2] \), then there is a corresponding decomposition of \( M' \) (i.e. one where \( C \) and \( C' \) have the same erasure) as \( C'[\text{let } x : \sigma' = M'_1 \text{ in } M'_2] \) where \( \sigma \) is more general than \( \sigma' \).

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

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

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

Rewriting rules used for constraint resolution can easily be adapted to retain the persist-
ent 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

\[
def x : \sigma \text{ in } R[x \leq \tau] \equiv \def x : \sigma \text{ in } R[x \leq \tau \land \sigma \leq \tau]
\]

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

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

\[
\lambda x. \lambda l_1. \lambda l_2. \text{let assoc} = \text{assoc } x \text{ in } (\text{assoc } l_1, \text{assoc } l_2)
\]

The principal type scheme \(\langle a \rangle\) is, by definition:

\[
\forall \alpha \exists \alpha_0 \alpha_1 \alpha_2 \beta. \begin{cases} 
\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\
def \Gamma \in \text{let assoc } x. \forall \gamma_1 \exists \gamma_2. (\text{assoc } \leq \gamma_2 \rightarrow \gamma_1), \gamma_1 \text{ in } \end{cases}, \alpha
\]

Since \(x : \alpha_9\) is in \(\Gamma\), the inner constraint can be resolved as follows:

\[
\exists \gamma_2. (\text{assoc } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2) \\
\equiv \exists \gamma_2. (\text{assoc } \leq \gamma_2 \rightarrow \gamma_1 \land x \leq \gamma_2 \land \alpha_0 \leq \gamma_2) \equiv \text{assoc } \leq \alpha_0 \rightarrow \gamma_1 \land x \leq \alpha_0
\]

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

\[
\text{assoc } \leq \alpha_0 \rightarrow \gamma_1 \land \forall \alpha \beta. \alpha \rightarrow \text{list } (\alpha \times \beta) \rightarrow \beta \leq \alpha_0 \rightarrow \gamma_1 \land x \leq \alpha_0 \\
\equiv \text{assoc } \leq \alpha_0 \rightarrow \gamma_1 \land \exists \alpha \beta. (\alpha = \alpha_0 \land \text{list } (\alpha \times \beta) \rightarrow \beta = \gamma_1) \land x \leq \alpha_0 \\
\equiv \exists \beta. (\text{assoc } \leq \alpha_0 \rightarrow \text{list } (\alpha_0 \times \beta) \rightarrow \beta \land \text{list } (\alpha_0 \times \beta) \rightarrow \beta = \gamma_1 \land x \leq \alpha_0)
\]
Hence, the type scheme of \( \text{assoc} \) is equivalent to
\[
\forall \beta \left[ \text{assoc} \leq \alpha_0 \rightarrow \text{list}(\alpha_0 \times \beta) \rightarrow \beta \land x \leq \alpha_0 \right]. \text{list}(\alpha_0 \times \beta) \rightarrow \beta
\]
and \( \{a_1\} \) is equivalent to:
\[
\forall \alpha \exists \alpha_0 \alpha_1 \alpha_2. \left( \begin{array}{l}
\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\
\text{def } \Gamma \text{ in } \text{let } \text{assocx}: \forall \beta \left[ \text{assoc} \leq \alpha_0 \rightarrow \text{list}(\alpha_0 \times \beta) \rightarrow \beta \land x \leq \alpha_0 \right]. \text{list}(\alpha_0 \times \beta) \rightarrow \beta \text{ in } \end{array} \right)
\]
Simplifying the remaining instantiation constraints in a similar way, we end up with the following resolved type scheme for \( \{a\} \):
\[
\forall \alpha_0 \beta_1 \beta_2 \left[ \begin{array}{l}
\alpha_0 \rightarrow \text{list}(\alpha_0 \times \beta_1) \rightarrow \text{list}(\alpha_0 \times \beta_2) \rightarrow \beta_1 \times \beta_2 \\
\text{def } \Gamma \text{ in } \text{let } \text{assocx}: \forall \gamma \left[ \text{assoc} \leq \alpha_0 \rightarrow \text{list}(\alpha_0 \times \gamma) \rightarrow \gamma \right]. \text{list}(\alpha_0 \times \gamma) \rightarrow \gamma \text{ in } \end{array} \right)
\]
This is a resolved form, from which we may build the elaboration of \( a_1 \):
\[
\Lambda \alpha_0 \beta_1 \beta_2 \lambda x: \alpha_0. \lambda l_1: \text{list}(\alpha_0 \times \beta_1). \lambda l_2: \text{list}(\alpha_0 \times \beta_2). \text{let } \text{assocx} = \Lambda \gamma. \text{assoc } \alpha_0 \gamma x \in (\text{assocx } \beta_1 \ l_1, \text{assocx } \beta_2 \ l_2)
\]
Type abstractions are determined by their corresponding type scheme in the resolved constraint; for instance, the type abstraction for the let-bound variable \( \text{assocx} \) is \( \gamma \) while the toplevel type abstraction is \( \alpha_0 \alpha_1 \beta_2 \). Type annotations on abstractions are determined by \( \Gamma \), which here contains \( x: \alpha_0; l_1: \text{list}(\alpha_0 \times \alpha_1); l_2: \text{list}(\alpha_0 \times \alpha_2) \). Type applications are inferred locally by looking at their corresponding type instantiations in the resolved constraints. For instance, we read from the constraint that \( \text{assocx} \) is let-bound with the type scheme \( \forall \gamma. \text{list}(\alpha_0 \times \gamma) \rightarrow \gamma \) (we dropped the constraint which is solved and equivalent to \text{true}) and that its \( i \)-th occurrence is used at type \( \text{list}(\alpha_0 \times \beta_i) \rightarrow \beta_i \). Matching the former against the latter gives the substitution \( \gamma \mapsto \beta_i \). Therefore, the type application for the \( i \)'s occurrence is be \( \beta_i \).

**Modular type reconstruction** One criticism of our approach is that the mechanism 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 $\alpha$ may be instantiated to any type that satisfies the constraint $C$ and not necessarily the most general one.

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

Exercise 36 (Minimal derivations) On the opposite, one may seek for less general typing derivations where all let-expressions are as instantiated as possible. Do such derivations exist? 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.

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.

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:

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

Typing and constraint generation are obvious:

$$ \frac{\Gamma \vdash a : \tau}{\Gamma \vdash (a : \tau) : \tau} \quad \llbracket (a : \tau) : \tau' \rrbracket = \llbracket a : \tau \rrbracket \land \tau = \tau' $$

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

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

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

\[
\lambda x. x : \alpha \to \alpha \quad (\lambda x. x + 1 : \alpha \to \alpha) \quad \text{let } f = (\lambda x. x : \alpha \to \alpha) \text{ in } (f 0, f \text{ true})
\]

Does it make sense, and is so, what does it mean? A short answer is that it does not mean anything, because \(\alpha\) is unbound. “There is no such thing as a free variable” (Alan Perlis).

A longer answer is that it is necessary to specify how and where variables are bound.

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

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

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

\[
a \ ::= \ldots \mid \exists \vec{\alpha}. a \mid \forall \vec{\alpha}. a
\]

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., \(\vec{\alpha} \not\# \Gamma, a \text{ in } \text{Gen}\)). The new (and updated) typing rules are as follows:

\[
\begin{align*}
\text{Exists} & \quad \Gamma \vdash [\vec{\alpha} \mapsto \vec{\tau}] a : \tau \\
\Gamma & \vdash \exists \vec{\alpha}. a : \tau \\
\text{Forall} & \quad \Gamma \vdash a : \tau \quad \vec{\alpha} \not\# \Gamma \\
\Gamma & \vdash \forall \vec{\alpha}. a : \forall \vec{\alpha}. \tau
\end{align*}
\]

As type annotations, the introduction of type variables are erased prior to runtime.

Exercise 38 Define the erasure of implicitly-typed terms and show that the erasure of a well-typed term is well-typed. Use this to justify the soundness of the extension of ML with type annotations with explicit introduction of type variables.

Constraint generation for the existential form is straightforward:

\[
\llangle (\exists \vec{\alpha}. a) : \tau \rrangle = \exists \vec{\alpha}. \langle a : \tau \rangle \quad \text{if } \vec{\alpha} \not\# \tau
\]

The type annotations inside \(a\) contain free occurrences of \(\vec{\alpha}\). Thus, the constraint \(\langle a : \tau \rangle\) contains such occurrences as well, which are bound by the existential quantifier.

For example, the expression \(\lambda x_1. \lambda x_2. \exists \alpha. ((x_1 : \alpha), (x_2 : \alpha))\) has principal type scheme \(\forall \alpha. \alpha \to \alpha \to \alpha \times \alpha\). Indeed, the generated constraint is of the form \(\exists \alpha. (\langle x_1 : \alpha \rangle \land \langle x_2 : \alpha \rangle \land \ldots)\), 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:
\[
\forall t, \phi[\alpha \mapsto t], \psi \vdash C \\
\phi, \psi \vdash \forall \alpha. C
\]

(To solve these constraints, we will use an extension of the unification algorithm called
unification under a mixed prefix—see §5.4.3.)

The need for universal quantification in constraints arises when polymorphism is required
by the programmer, as opposed to inferred by the system. Constraint generation for the
universal form is somewhat subtle. A naive definition fails:
\[
\{\forall \bar{\alpha}. a : \tau\} = \forall \bar{\alpha}. \{a : \tau\} \quad \text{if } \bar{\alpha} \neq \tau \quad \text{Wrong!}
\]
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:
\[
\{\forall \alpha. (\lambda x. x : \alpha \to \alpha) : \text{int} \to \text{int}\} \\
\equiv \forall \alpha. \{\{\lambda x. x : \alpha \to \alpha\} : \text{int} \to \text{int}\} \\
\equiv \forall \alpha. (\{\lambda x. x : \alpha \to \alpha\} \land \alpha = \text{int}) \equiv \forall \alpha. (\text{true} \land \alpha = \text{int}) \equiv \text{false}
\]
A correct definition is:
\[
\{\forall \bar{\alpha}. a : \tau\} = \forall \bar{\alpha}. \exists \gamma. \{a : \gamma\} \land \exists \bar{\alpha}. \{a : \tau\}
\]
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 \cite[p. 112]{Pottier:2003}. The solution defines:
\[
\{\forall \bar{\alpha}. a : \tau\} = \text{let } x : \forall \bar{\alpha}, \beta[\{a : \beta\}], \beta \text{ in } x \leq \tau
\]
where the new constrain form satisfies the equivalence:
\[
\text{let } x : \forall \bar{\alpha}, \beta[C_1]. \tau \text{ in } C_2 \equiv \forall \bar{\alpha}. \exists \beta. C_1 \land \text{def } x : \forall \bar{\alpha}, \beta[C_1]. \tau \text{ in } C_2
\]
Annotating a term with a type scheme, rather than just a type, is now just syntactic sugar:
\[
(a : \forall \bar{\alpha}. \tau) \triangleq \forall \bar{\alpha}. (a : \tau) \quad \text{if } \bar{\alpha} \neq a
\]
In that particular case, constraint generation is in fact simpler:
\[
\{\{a : \forall \bar{\alpha}. \tau\} : \tau'\} \equiv \forall \bar{\alpha}. \{a : \tau\} \land (\forall \bar{\alpha}. \tau) \leq \tau'
\]
Exercise 39  Check this equivalence.

Examples  Consider the following two examples:

\[
\begin{align*}
\langle(\exists \alpha. (\lambda x. x + 1 : \alpha \to \alpha)) : \text{int} \to \text{int}\rangle & \quad \langle(\forall \alpha. (\lambda x. x + 1 : \alpha \to \alpha)) : \text{int} \to \text{int}\rangle \\
\Rightarrow & \quad \exists \alpha.\langle(\lambda x. x + 1 : \alpha \to \alpha) : \text{int} \to \text{int}\rangle \quad \vdash \quad \forall \alpha.\exists \gamma.\langle(\lambda x. x + 1 : \alpha \to \alpha) : \gamma\rangle \\
\Rightarrow & \quad \exists \alpha. (\alpha = \text{int}) \quad \vdash \quad \forall \alpha.\exists \gamma. (\alpha =\text{int} \land \alpha \to \alpha = \gamma) \\
\Rightarrow & \quad \text{true} \quad \vdash \quad \forall \alpha.\alpha = \text{int} \\
\Rightarrow & \quad \text{false}
\end{align*}
\]

The left-hand side example is well-typed: The system infers that \(\alpha\) must be \text{int}. Because \(\alpha\) is a local type variable, it does not appear in the final constraint. The right-hand side example is ill-typed: The system checks that \(\alpha\) is used in an abstract way, which is not the case here, since the code implicitly assumes that \(\alpha\) is \text{int}. By contrast, the following example is well-typed:

\[
\begin{align*}
\langle(\forall \alpha. (\lambda x. x : \alpha \to \alpha)) : \text{int} \to \text{int}\rangle & \quad \vdash \quad \forall \alpha.\exists \gamma.\langle(\lambda x. x : \alpha \to \alpha) : \text{int} \to \text{int}\rangle \\
\Rightarrow & \quad \forall \alpha.\exists \gamma. \alpha \to \alpha = \gamma \land \exists \alpha.\alpha = \text{int} \\
\Rightarrow & \quad \text{true}
\end{align*}
\]

The system checks that \(\alpha\) is used in an abstract way, which is indeed the case here. It also checks that, if \(\alpha\) is appropriately instantiated, the code admits the expected type \text{int} \to \text{int}.

The two next examples are similar and show the importance of the scope of existential variables. In the first one, the variable \(\alpha\) is bound outside the let construct:

\[
\begin{align*}
\langle(\exists \alpha. (\text{let } f = (\lambda x. x : \alpha \to \alpha) \text{ in } (f 0, f \text{ true})) : \gamma)\rangle & \quad \exists \alpha. (\text{let } f : \alpha \to \alpha \text{ in } \exists \gamma_1 \gamma_2. (f \leq \text{int} \to \gamma_1 \land f \leq \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma)) \\
\Rightarrow & \quad \exists \alpha. (\alpha = \text{int} \land \alpha \to \alpha = \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma) \\
\Rightarrow & \quad \exists \alpha. (\alpha = \text{int} \land \alpha = \text{bool}) \\
\Rightarrow & \quad \text{false}
\end{align*}
\]

Then \(f\) receives the monotype \(\alpha \to \alpha\) and the example is ill-typed. In the other example, \(\alpha\) is bound within the let construct:

\[
\begin{align*}
\langle(\text{let } f = \exists \alpha. (\lambda x. x : \alpha \to \alpha) \text{ in } (f 0, f \text{ true})) : \gamma)\rangle & \quad \text{let } f : \forall \beta[\exists \alpha. (\alpha \to \alpha = \beta)], \beta \in \exists \gamma_1 \gamma_2. (f \leq \text{int} \to \gamma_1 \land f \leq \text{bool} \to \gamma_2 \land \gamma_1 \times \gamma_2 = \gamma) \\
\Rightarrow & \quad \text{let } f : \forall \alpha. \alpha \to \alpha \text{ in } \exists \gamma_1 \gamma_2. (\ldots) \\
\Rightarrow & \quad \exists \gamma_1 \gamma_2. (\gamma_1 \times \text{false} = \gamma_2 \times \gamma_1 \times \gamma_2 = \gamma) \\
\Rightarrow & \quad \text{int} \times \text{bool} = \gamma
\end{align*}
\]

Here, the term \(\exists \alpha. (\lambda x. x : \alpha \to \alpha)\) has the same principal type scheme as \(\lambda x. x\), namely \(\forall \alpha. \alpha \to \alpha\), which is the type scheme that \(f\) receives.
5.4. TYPE ANNOTATIONS

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:

\[
\begin{align*}
\text{FixAbs} & : \frac{\Gamma, f : \tau \vdash \lambda x. a : \tau}{\Gamma \vdash \mu f. \lambda x. a : \tau} \\
\text{LetRec} & : \frac{\Gamma, f : \tau_1 \vdash \lambda x. a_1 : \tau_1 \quad \bar{\alpha} \not\# \Gamma, a_1 \quad \Gamma, f : \forall \bar{\alpha}. \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau_2}
\end{align*}
\]

These rules require occurrences of \( f \) to have monomorphic type within the recursive definition (that is, within \( \lambda x. a_1 \)). This is visible also in terms of type inference, as the two following constraints are equivalent:

\[
\langle \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau \rangle \equiv \text{let } f : \forall \alpha \beta [ \langle \text{let } f : \alpha \to \beta; x : \alpha \text{ in } \langle a_1 : \beta \rangle \rangle, \alpha \to \beta \text{ in } \langle a_2 : \tau \rangle
\]

On the right-hand side, all occurrences of \( f \) within \( a_1 \) have the same type \( \alpha \to \beta \). This is problematic in some situations, most particularly when defining functions over nested algebraic data types [Bird and Meertens (1998; Okasaki (1999)].

This problem is solved by introducing polymorphic recursion, that is, by allowing \( \mu \)-bound variables to receive a polymorphic type scheme, using the following typing rules:

\[
\begin{align*}
\text{FixAbsPoly} & : \frac{\Gamma, f : \sigma \vdash \lambda x. a : \sigma}{\Gamma \vdash \mu f. \lambda x. a : \sigma} \\
\text{LetRecPoly} & : \frac{\Gamma, f : \sigma \vdash \lambda x. a_1 : \sigma \quad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \text{let rec } f \ x = a_1 \text{ in } a_2 : \tau}
\end{align*}
\]

This extension of ML is due to Mycroft (1984).

In System F, there is no problem to begin with; no extension is necessary. Polymorphic recursion alters, to some extent, Damas and Milner’s type system. Now, not only let-bound, but also \( \mu \)-bound variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed \( \lambda \)-calculus. This has two noticeable consequences: monomorphization, a technique employed in some ML compilers [Tolmach and Oliva (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:

\[
\begin{align*}
\text{FixAbsPoly} & : \Gamma, f : \sigma \vdash \lambda x. a : \sigma \\
\text{LetRecPoly} & : \Gamma, f : \sigma \vdash \lambda x. a : \sigma \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \mu (f : \sigma). \lambda x. a : \sigma & \\
\Gamma \vdash \text{let rec } (f : \sigma) = \lambda x. a_1 \text{ in } a_2 : \tau & \\
\end{align*}
\]

The type scheme \(\sigma\) no longer has to be guessed. With this feature, contrary to what was said earlier (p. 109), type annotations are not just restrictive: they are sometimes required for type inference to succeed. The constraint generation rule becomes:

\[
\langle \text{let rec } (f : \sigma) = \lambda x. a_1 \text{ in } a_2 : \tau \rangle = \text{let } f : \sigma \text{ in } (\langle \lambda x. a_1 : \sigma \rangle \land \langle a_2 : \tau \rangle)
\]

It is clear that \(f\) receives type scheme \(\sigma\) both inside and outside of the recursive definition.

5.4.3 Unification under a mixed prefix

Unification under a mixed prefix means unification in the presence of both existential and universal quantifiers. We extend the basic unification algorithm with support for universal quantification. The solved forms are unchanged: universal quantifiers are always eliminated.

In short, in order to reduce \(\forall \alpha. C\) to a solved form, where \(C\) is itself a solved form—see (Pottier and Rémy, 2003, p. 109) for details:

- If a rigid variable is equated with a constructed type, fail.
  For example, \(\forall \alpha. \exists \beta. (\alpha = \beta \rightarrow \gamma)\) is false.

- If two rigid variables are equated, fail.
  For example, \(\forall \alpha \beta. (\alpha = \beta)\) is false.

- If a free variable dominates a rigid variable, fail.
  For example, \(\forall \alpha. \exists \beta. (\gamma = \alpha \rightarrow \beta)\) is false.

- Otherwise, one can decompose \(C\) as \(\exists \beta. (C_1 \land C_2)\), where \(\alpha \beta \neq C_1\) and \(\exists \beta. C_2 \equiv \text{true}\); in that case, \(\forall \alpha. C\) reduces to just \(C_1\).
  For example, \(\forall \alpha. \exists \beta. (\beta = \alpha \rightarrow \gamma \land \gamma = \gamma_1 \rightarrow \gamma_2)\) reduces to just \(\exists \gamma_1 \gamma_2. (\gamma = \gamma_1 \rightarrow \gamma_2)\).
  The constraint \(\forall \alpha. \exists \beta. (\beta = \alpha \rightarrow \gamma)\) is equivalent to \(\text{true}\).

OCaml implements a form of unification under a mixed prefix. This is illustrated by the following interactive OCaml session:

```
let module M : sig val id : 'a -> 'a end = struct let id x = x + 1 end in M.id
```

Values do not match: val id : int -> int
is not included in val id : 'a -> 'a

This gives rise to a constraint of the form \(\forall \alpha. \alpha = \text{int}\), while the following example gives rise to a constraint of the form \(\exists \beta. \forall \alpha. \alpha = \beta\):
5.5. EQUI- AND ISO-RECURSIVE TYPES

```ocaml
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 \( \tau ::\ =\ \text{unit} | \tau \times \tau | \tau + \tau \), then it is clear that every type describes a finite set of values. For every \( k \), the type of lists of length at most \( k \) is expressible using this grammar. However, the type of lists of unbounded length is not: “A list is either empty or a pair of an element and a list.” We need something like this:

\[
\text{list}\ \alpha \odot \text{unit} + \alpha \times \text{list}\ \alpha
\]

But what does \( \odot \) 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:

\[
\tau ::= \alpha | F\ \bar{\tau}
\]

is no longer interpreted inductively. Instead, types are the regular trees built on top of this signature. If desired, it is possible to use finite syntax for recursive types:

\[
\tau ::= \alpha | \mu\alpha.(F\ \bar{\tau})
\]

We do not allow the seemingly more general \( \mu\alpha.\tau \), because \( \mu\alpha.\alpha \) is meaningless, and \( \mu\alpha.\beta \) or \( \mu\alpha.\mu\beta.\tau \) are useless. If we write \( \mu\alpha.\tau \), it should be understood that \( \tau \) is contractive, that is, \( \tau \) is a type constructor application. For instance, the type of lists of elements of type \( \alpha \) is:

\[
\mu\beta.(\text{unit} + \alpha \times \beta)
\]

Each type in this syntax denotes a unique regular tree, sometimes known as its infinite unfolding. Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to decide whether two types are equal, that is, have identical infinite unfoldings.
This can be done efficiently, either via the algorithm for comparing two DFAs, or by uni-
fication. (The latter approach is simpler, faster, and extends to the type inference problem.)

One can also prove [Brandt and Henglein (1998)] that equality is the least congruence

generated by the following two rules:

\[
\begin{align*}
\text{Fold/Unfold} \\
\mu \alpha. \tau = [\alpha \mapsto \mu \alpha. \tau] \tau \\
\text{Uniqueness} \\
\tau_1 = [\alpha \mapsto \tau_1] \tau_2 = [\alpha \mapsto \tau_2] \tau \\
\tau_1 = \tau_2
\end{align*}
\]

In both rules, \( \tau \) must be contractive. This axiomatization does not directly lead to an efficient
algorithm for deciding equality, though. In the presence of equi-recursive types, structural
induction on types is no longer permitted—but we never used it anyway. It remains true that
\( F \tilde{\tau}_1 = F \tilde{\tau}_2 \) implies \( \tilde{\tau}_1 = \tilde{\tau}_2 \)—this was used in our Subject Reduction proofs. It remains true
that \( F_1 \tilde{\tau}_1 = F_2 \tilde{\tau}_2 \) implies \( F_1 = F_2 \)—this was used in our Progress proofs. So, the reasoning
that leads to type soundness is unaffected.

**Exercise 40** Prove type soundness for the simply-typed \( \lambda \)-calculus in Coq. Then, change
the syntax of types from \texttt{Inductive} to \texttt{CoInductive}.

How is type inference adapted for equi-recursive types? The syntax of constraints is
unchanged: they remain systems of equations between finite first-order types, without \( \mu \)'s. Their interpretation changes: they are now interpreted in a universe of regular trees. As
a result, constraint generation is unchanged; constraint solving is adapted by removing the
occurs check.

**Exercise 41** Describe solved forms and show that every solved form is either false or satis-
fiable.

Here is a function that measures the length of a list:

\[
\mu (\text{length}) . \lambda x . \text{case } x \text{ of } \lambda() . 0 \diamond \lambda(y, z) . (1 + \text{length } z)
\]

Type inference gives rise to the cyclic equation \( \beta = \text{unit} + \alpha \times \beta \), where \text{length} has type \( \beta \rightarrow \text{int} \). That is, \text{length} has principal type scheme: \( \forall \alpha . (\mu \beta. \text{unit} + \alpha \times \beta) \rightarrow \text{int} \) or, equivalently, principal
constrained type scheme: \( \forall \alpha[\beta = \text{unit} + \alpha \times \beta] \). \( \beta \rightarrow \text{int} \). The cyclic equation that characterizes
lists was never provided by the programmer, but was inferred.

OCaml implements equi-recursive types upon explicit request, launching the interactive
session with the command \texttt{"ocaml -rectypes"}:

\[
\text{type } ('a, 'b) \text{ sum } = \text{Left of } 'a | \text{Right of } 'b \\
\text{type } ('a, 'b) \text{ sum } = \text{Left of } 'a | \text{Right of } 'b \\
\text{let rec } \text{length } x = \text{function } \text{Left } () \rightarrow 0 | \text{Right } (y, z) \rightarrow 1 + \text{length } z \\
\text{val } \text{length } : ((\text{unit, 'b } * 'a) \text{ sum as } 'a) \rightarrow \text{int } = (\text{fun})
\]
5.5. EQUI- AND ISO-RECURSIVE TYPES

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:

\[
\text{let rec } \text{map } f = \text{function } [\] \to [\] | y :: z \to \text{map } f y :: \text{map } f z
\]

\[
\text{val map : 'a } \to ('b \text{ list as 'b }) \to ('c \text{ list as 'c }) = \langle \text{fun} \rangle
\]

\[
\text{map } (\text{fun } x \to x + 1) \begin{array}{l} [1; 2] \end{array}
\]

This expression has type int but is used with type 'a list as 'a

\[
\text{map } () \begin{array}{l} [\[]; [\[]] \end{array}
\]

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:

\[
\text{let } f \ x = x\#\text{hello } x;\]

\[
\text{val f : (} < \text{hello : 'a } \to 'b; \ldots > \text{ as 'a) } \to 'b = \langle \text{fun} \rangle
\]

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:

\[
D\bar{\alpha} \approx \tau \quad \text{(where ftv}(\tau) \subseteq \bar{\alpha})
\]

Each such declaration adds a unary constructor \(\text{fold}_D\) and a unary destructor \(\text{unfold}_D\) with the following types and the new reduction rule:

\[
\text{fold}_D : \forall \bar{\alpha}. \tau \to D\bar{\alpha} \quad \text{unfold}_D : \forall \bar{\alpha}. D\bar{\alpha} \to \tau \quad \text{unfold}_D (\text{fold}_D v) \longrightarrow v
\]

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 \(\text{fold}_D\) and \(\text{unfold}_D\) with the semantics of the identity function. Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the [UNIQUENESS] typing rule.
For example, iso-recursive lists can be defined as follows. A parametrized, iso-recursive type of lists is: $\text{list } \alpha \approx \text{unit} + \alpha \times \text{list } \alpha$. The empty list is: $\text{fold}_{\text{list}} (\text{inj}_1 () : \forall \alpha. \text{list } \alpha$. A function that measures the length of a list is:

$$\mu (\text{length}) \cdot \lambda x . \text{case } (\text{unfold}_{\text{list}} x) \text{ of } \lambda(). 0 \circ \lambda (x, x s). 1 + \text{length } x s : \forall \alpha. \text{list } \alpha \to \text{int}$$

One folds upon construction and unfolds upon deconstruction.

In the iso-recursive approach, types remain finite. The type $\text{list } \alpha$ is just an application of a type constructor to a type variable. As a result, type inference is unaffected. The occurs check remains.

### 5.5.3 Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums. This suppresses the verbosity of explicit folds and unfolds as well as the fragility and inconvenience of numeric indices—instead, named record fields and data constructors are used. For instance, $\text{fold}_{\text{list}} (\text{inj}_1 ())$ is replaced with $\text{Nil } ()$

An algebraic data type constructor $D$ is introduced via a record type or variant type definition:

$$D \vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_{\ell} \quad \text{or} \quad D \vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_{\ell}$$

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:

$$C := \ldots | (\ell_1 = \cdot \ldots \ell_n = \cdot) \quad d := \ldots | (\cdot \ell_1) \ldots (\cdot \ell_n)$$

$$\{\ell_1 = \ldots, \ell_n = \cdot\} : \forall \vec{\alpha}. \tau_{\ell_1} \to \ldots \tau_{\ell_n} \to D \vec{\alpha} \quad \ell : \forall \vec{\alpha}. D \vec{\alpha} \to \tau_{\ell}$$

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:

$$C := \ldots | (\ell) \quad d := \ldots | \text{case } \cdot \text{ of } [\ell_1 : \ldots \ell_n : \cdot] \quad \ell : \forall \vec{\alpha}. \tau_{\ell} \to D \vec{\alpha}$$

$$\text{case } \cdot \text{ of } [\ell_1 : \ldots \ell_n : \cdot] : \forall \vec{\alpha} \beta. D \vec{\alpha} \to (\tau_{\ell_1} \to \beta) \to \ldots (\tau_{\ell_n} \to \beta) \to \beta$$

For example, an algebraic data type of lists is $\text{list } \alpha \approx \text{ Nil : unit} + \text{ Cons : } \alpha \times \text{ list } \alpha$ gives rise to:

$$\text{case } \cdot \text{ of } [\text{Nil : } \ldots \text{ Cons : } \cdot] : \forall \alpha \beta. \alpha \to (\text{unit } \to \beta) \to ((\alpha \times \text{ list } \alpha) \to \beta) \to \beta$$

$$\text{Nil : } \forall \alpha. \text{ unit } \to \text{ list } \alpha$$

$$\text{Cons : } \forall \alpha. (\alpha \times \text{ list } \alpha) \to \text{ list } \alpha$$
A function that measures the length of a list is:

\[
\mu(length).\lambda x.\text{case } x\text{ of } \text{Nil} : \lambda()., \text{Cons} : \lambda(y, z).1 + length z : \forall \alpha.\text{list } \alpha \rightarrow \text{int}
\]

**Mutable record fields** In OCaml, a record field can be marked *mutable*. This introduces an extra binary destructor for writing this field: \((\cdot . \ell \leftarrow \cdot)\) of type \(\forall \alpha. D \rightarrow \tau_\ell \rightarrow \text{unit}\). However, this also makes record construction a destructor since, when fully applied it is *not* a value but it allocates a piece of store and returns its location. Thus, due to the value restriction, the type of such expressions cannot be generalized.

## 5.6 HM(X)

Soundness and completeness of type inference are in fact easier to prove if one adopts a *constraint-based specification* of the type system, as in the language HM(X) introduced by Odersky et al. (1999).

In HM(X), judgments take the form \(C, \Gamma \vdash a : \tau\), called a constrained typing judgments. Read under the assumption \(C\) and typing environment \(\Gamma\), the program \(a\) has type \(\tau\). Here \(C\) constrains free type variables of the judgment while \(\Gamma\) provides the type of free program variables of \(a\). The constraint \(C\) ranges over first-order typing constraints—except that we require type constraints to have no free program variables. In a constrained typing judgment \(C, \Gamma \vdash a : \tau\),

The parameter \(X\) in HM(X) stands for the logic of the constraint language. We have so far only consider constraints with an equality predicate. However, the equality replaced may be by an asymmetric subtyping predicate \(\leq\), which makes the language of constraints richer.

The typing rules also use an entailment predicate \(C \vdash C'\) between constraints that is more general than constraint equivalence. Entailment is defined as expected: \(C \vdash C'\) if and only if any ground assignment that satisfies \(C\) also satisfies \(C'\).
Typing rules for \( \text{HM}(X) \) are presented in Figure 5.7. Moreover, judgment are taken up to constraint equivalence. The constraint \( \exists \sigma \) in the premise of Rule \( \text{HM-VAR} \) is an abbreviation for \( \exists \alpha. C_0 \) where \( \sigma = \forall \alpha [C_0]. \tau \). A valid judgment is one that has a derivation with those typing rules. In a valid judgment, \( C \) may not be satisfiable. A program is well-typed in environment \( \Gamma \) if it has a valid judgment \( C, \Gamma \vdash \alpha : \tau \) for some \( \tau \) and satisfiable constraint \( C \).

When considering equality only constraints, \( \text{HM}(=) \) is in fact equivalent to \( \text{ML} \): if \( \Gamma \) and \( \tau \) contain only Damas-Milner’s type schemes, then \( \Gamma \vdash \alpha : \tau \) in \( \text{ML} \) if and only if \( \text{true}, \Gamma \vdash \alpha : \tau \) in \( \text{HM}(X) \). Moreover, if \( \text{C}, \Gamma \vdash \alpha : \tau \) in \( \text{HM}(X) \) and \( \varphi \) is an idempotent solution of \( C \), we have \( \text{true}, \Gamma \vdash \alpha : \tau \varphi \) in \( \text{HM}(X) \) where \( (\cdot)\varphi \) translates \( \text{HM}(X) \) type schemes into \( \text{ML} \) type schemes—applying the substitution \( \varphi \) on the fly.

As for \( \text{ML} \), there is an equivalent syntax-directed presentation of the typing rules. However, we may take advantage of program variables in constraints to go one step further and mix the constraint \( C \) (without free program variables) and the typing environment \( \Gamma \) into a single constraint \( C \) now with possibly free program variables. Judgments take the form \( C \vdash \alpha : \tau \) where \( C \) constrains type variables and assign constrained type schemes to program variables. The type system, called \( \text{PCB}(X) \), is described on Figure 5.8. It is equivalent to \( \text{HM}(X) \)—see (Pottier and Rémy, 2005) for the precise comparison.

For example of a derivation in \( \text{PCB}(X) \), let \( a \) be \( \text{let } y = \lambda x. x \text{ in } y y \):

\[
\begin{align*}
\text{VAR} & \quad x \leq \alpha \vdash x : \alpha \\
\text{FUN} & \quad \text{let } x : \alpha_0 \text{ in } x \leq \alpha \vdash \lambda x. x : \alpha_0 \rightarrow \alpha \\
\text{LET} & \quad \text{let } y : \alpha_0 \text{ in } x \leq \alpha \vdash \lambda x. x : \alpha_0 \rightarrow \alpha \\
\text{VAR} & \quad y \leq \beta_2 \vdash y : \beta_2 \\
\text{VAR} & \quad y \leq \beta_2 \vdash y \vdash y : \beta_1 \\
\text{EXISTS} & \quad \exists \beta_2. C \vdash \alpha : \beta_1 \\
\end{align*}
\]

where \( C \) is

\[
\text{let } y : \forall \alpha_0 [\text{let } x : \alpha_0 \text{ in } x \leq \alpha], \alpha_0 \rightarrow \alpha \text{ in } y \leq \beta_2 \rightarrow \beta_1 \land y \leq \beta_2
\]

The constraint \( C \) can be simplified as follows:

\[
\exists \beta_2. C = \exists \beta_2, \text{let } y : \forall \alpha_0 [\alpha_0 = \alpha], \alpha_0 \rightarrow \alpha \text{ in } y \leq \beta_2 \rightarrow \beta_1 \land y \leq \beta_2
\]

Hence, we also have \( \exists \alpha. \beta_1 = \alpha \rightarrow \alpha \vdash \alpha : \beta_1 \). This is a valid judgment, but not a satisfiable one. However, by rule \( \text{Pcb-Sub} \) and \( \text{Pcb-Exists} \) we have \( \exists \beta_1. (\exists \alpha. \beta_1 \alpha \rightarrow \alpha) \land \beta_1 = \beta \rightarrow \beta \vdash a : \beta \rightarrow \beta \), which is equivalent to \( \text{true}, a : \beta \rightarrow \beta \). The type inference algorithm for \( \text{ML} \) is sound and complete for \( \text{PCB}(X) \):

- **Soundness**: \( \ll a : \tau \gg \vdash a : \tau \). The constraint inferred for a typing validates the typing.

- **Completeness**: If \( C \vdash a : \tau \) then \( C \vdash \ll a : \tau \gg \). The constraint inferred for a typing is more general than any constraint that validates the typing.
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).
### Chapter 5. Type Reconstruction

#### 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:

\[ M ::= x \mid \lambda x: \tau. M \mid M \cdot M \mid \Lambda \alpha. M \mid M \cdot \tau \mid \lambda x: ?. M \mid ? \mid \text{let } f = \Lambda^2 \alpha_1 \ldots \Lambda^2 \alpha_n. M \text{ in } M \]

The new let-binding form is used to declare type arguments that will be made implicit. Then, every occurrence of such a variable automatically adds type-application holes at the corresponding positions and type parameters will be inferred using second-order unification. This amounts to understanding the new let-binding form as follows:

\[ \text{let } f = \Lambda^2 \alpha_1 \ldots \Lambda^2 \alpha_n. M_1 \text{ in } M_2 \triangleq \text{let } f = \Lambda \alpha_1 \ldots \Lambda \alpha_n. M_1 \text{ in } [f \mapsto f ? \ldots ?]M_2 \]

Type inference in this language still reduces to second-order unification.

#### 5.7.2 Bidirectional type inference

Type-checking in explicit simply-typed 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, \( \Gamma \) and \( a \) would be inputs and \( \tau \) is an output in the judgment \( \Gamma \vdash a : \tau \), which we may represent as \( \Gamma^! \vdash a^! : \tau^! \). However, in the rule for abstraction:

\[
\text{Abs} \\
\frac{\text{\( \Gamma, x : \tau_0 \vdash a : \tau \)}}{\text{\( \Gamma \vdash \lambda x. a : \tau_0 \rightarrow \tau \)}}
\]

the type \( \tau_0 \) is used both as input (in the premise) and as an output in the conclusion. Hence, type-checking the implicit simply-typed 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
judgments $\Gamma \vdash a \downarrow \tau$ to check that $a$ may be assigned the type $\tau$ and $\Gamma \vdash a \uparrow \tau$ to infer the type $\tau$ of $a$ (or with information flows $\Gamma \uparrow a \downarrow \tau^I$ and $\Gamma \downarrow a \uparrow \tau^I$). Both judgments are recursively defined by the rules of Figure ??: the checking mode can call the inference mode when needed; conversely, annotations may be used to turn inference mode into checking mode. (As a particular case, annotations on type abstractions enable the inference mode.)

An example of bidirectional derivation is given on Figure 5.10. The type $\tau$ stands for $(\tau_1 \to \tau_1) \to \tau_2$ and the environment $\Gamma$ is $f : \tau$.

The bidirectional method can be extended to deal with polymorphic types, but it is more complicated. The idea, due to Cardelli (1993), was popularized by Pierce and Turner (2000), and Odersky et al. (2001) and is still being improved Dunfield (2009).

**Predicative polymorphism** 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 \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \) 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 \( \text{VAR} \). Hence, \( \phi \tau = \phi \Gamma(x) \). By definition of satisfiability this implies \( \phi \vdash \tau = \Gamma(x) \). By definition of typing constraint, this is \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \).

Conversely, assume \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \). By definition of typing constraint, this is \( \phi \vdash \tau = \Gamma(x) \). By inversion of satisfiability we must have \( \phi \tau = \phi \Gamma(x) \). Hence, by rule \( \text{VAR} \) we have \( \phi \Gamma \vdash a : \phi \tau \).

Case \( a \) is \( a_1 a_2 \): Assume \( \phi \Gamma \vdash a : \phi \tau \). By rule \( \text{APP} \) there exists \( \tau_2 \) such that \( \phi \Gamma \vdash a_1 : \tau_2 \rightarrow \phi \tau \) and \( \phi \Gamma \vdash a_2 : \tau_2 \). Let \( \beta \neq \Gamma \) and \( \phi' \) be \( \phi, \beta \mapsto \tau_2 \). We have \( \phi '\Gamma \vdash a_1 : \phi' \beta \rightarrow \tau \) and \( \phi '\Gamma \vdash a_2 : \beta \). Hence, by induction hypothesis \( \phi' \vdash \langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle \) and \( \phi' \vdash \langle \Gamma \vdash a_2 : \beta \rangle \). Thus, \( \phi \vdash \exists \beta. \langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle \land \langle \Gamma \vdash a_2 : \beta \rangle \). i.e. \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \).

Conversely, assume \( \phi \vdash \langle \Gamma \vdash a : \tau \rangle \). We have \( \phi \vdash \exists \beta. \langle \Gamma \vdash a_2 : \beta \rangle \land \langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle \). We may assume w.l.o.g. that \( \beta \neq \phi \). There must exist \( \phi' \) of the form \( \phi, \beta \mapsto \tau_2 \) such that \( \phi' \vdash \langle \Gamma \vdash a_2 : \beta \rangle \land \langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle \). By induction hypothesis, this implies \( \phi \Gamma \vdash a_2 : \phi' \beta \) and \( \phi \Gamma \vdash a_1 : \phi' \beta \rightarrow \tau \), i.e. \( \phi \Gamma \vdash a_2 : \tau_2 \) and \( \phi \Gamma \vdash a_1 : \phi \tau_2 \rightarrow \tau \). By rule \( \text{APP} \) we have \( \phi \Gamma \vdash a_1 a_2 : \phi \tau \).
5.8. PROOFS AND SOLUTION TO EXERCISES

Case a is \( \lambda x. a_1 \): Assume \( \phi \Gamma \vdash a : \phi \tau \). We may assume \( w.l.o.g. \) that \( x \not\in \Gamma \). By rule \textbf{Fun} there must exist \( \tau_1 \) and \( \tau_2 \) such that \( \phi \Gamma, x : \tau_2 \vdash a_1 : \tau_1 \) and \( \phi \tau = \tau_2 \rightarrow \tau_1 \). Let \( \beta_1 \) and \( \beta_2 \) be disjoint from \( \Gamma \) and \( \phi' \) be \( \phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1 \). Then, both \( \phi'((\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1) \) and \( \phi'\tau = \phi'(\beta_2 \rightarrow \beta_1) \) hold. By induction hypothesis, \( \phi \vdash \exists \beta_1 \beta_2. (\Gamma, x : \beta_2 \vdash a_1 : \beta_1) \wedge \tau = \beta_2 \rightarrow \beta_1 \). That is, \( \phi \vdash (\Gamma \vdash a : \tau) \).

Conversely, assume \( \phi \vdash (\Gamma \vdash a : \tau) \). By definition of constraints, we have \( \phi \vdash \exists \beta_1 \beta_2. (\Gamma, x : \beta_2 \vdash a_1 : \beta_1) \wedge \tau = \beta_2 \rightarrow \beta_1 \) for some \( x \) disjoint from \( \Gamma \). We may assume \( w.l.o.g. \) that \( \beta_1, \beta_2 \not\in \phi \). There must exist \( \phi' \) of the form \( \phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1 \) such that \( \phi' \vdash (\Gamma, x : \beta_2 \vdash a_1 : \tau_1) \) and \( \phi'\tau = \beta_2 \rightarrow \beta_1 \). By induction hypothesis, \( \phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1 \) and \( \phi'\tau = \phi'(\beta_2 \rightarrow \beta_1) \). That is, \( \phi\Gamma, x : \tau_2 \vdash a_1 : \tau_1 \) and \( \phi \tau = \tau_2 \rightarrow \tau_1 \). Hence, by rule \textbf{Fun} we have \( \phi \Gamma \vdash a : \phi \tau \).

\textbf{Solution of Exercise 36}

See Bjørner (1994).

\textbf{Solution of Exercise 37}

Consider the module \texttt{struct f = let f = \( \lambda x. x \) in f f end}. In core ML, the expression has principal type \( \alpha \rightarrow \alpha \)—but \( \alpha \) cannot be generalized. Hence, \texttt{sig f : \forall \alpha. \alpha \rightarrow \alpha end} is not a signature for this module; nor is \texttt{sig f : \alpha \rightarrow \alpha end} since it is not a well-formed one. Correct signatures are \texttt{sig f : \tau \rightarrow \tau end} for any \( \tau \), but they do not have a best element.
Bibliography

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


