Type systems for programming languages

Didier Rémy

Academic year 2014-2015
Version of November 21, 2014

Contents

Chapter 5

Type reconstruction

5.1 Introduction

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

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

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

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

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

This way, types need never be guessed, just looked up into the typing context, instantiated, or checked for equality. This is exactly the situation with explicitly-typed presentations of the typing rules. This is also the traditional approach of Pascal, C, C++, Java, *etc.*: formal procedure parameters, as well as local variables, are assigned explicit types. The types of expressions are synthesized bottom-up.

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

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

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

5.2 Type inference for simply-typed λ-calculus

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

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

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

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

5.2.1 Constraints

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

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

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

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

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

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

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

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

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

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

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

5.2.2 A detailed example

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

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

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

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

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

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

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

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

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

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

By successive elimination of α_4 then α_2, we get:

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

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

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

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

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

Applying this simplification under a context, with the rule:

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

we have:

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

We may simplify the right-hand application analogously.

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

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

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

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

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

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

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

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

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

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

This technique is used also by Standard ML and Haskell.

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

One can check the following equivalence:

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

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

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

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

5.2.3 Soundness and completeness of type inference

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

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

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

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

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

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

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

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

5.2.4 Constraint solving

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

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

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

Figure 5.2: Solving unification constraints

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Type inference for ML

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

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

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

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

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

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

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

Figure 5.3: Type inference algorithm for ML

5.3.2 Constraint-based type inference for ML

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

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

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

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

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

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

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

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

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

Figure 5.4: Constraints with program variables

parameter Γ. Constraint generation is defined in Figure 5.4

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

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

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

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

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

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

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

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

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

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

The interpretation of constraints must be redefined since the environment ψ now maps program variables to sets of ground types. The environment ϕ still maps type variables to ground types. Hence, a type variable α still denotes a ground type. A variable x now denotes a *set* of ground types. Instantiation constraints are interpreted as *set membership.* The rules for the new form of constraints are:

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

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

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

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

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

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

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

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

$$\mathsf{let}\ x : \sigma\ \mathsf{in}\ C \;\triangleq\; \mathsf{def}\ x : \sigma\ \mathsf{in}\ ((\exists\alpha.x \leq \alpha) \wedge C)$$

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

$$\mathsf{let}\ x : \forall\bar{\alpha}[C].\tau\ \mathsf{in}\ C' \;\triangleq\; \exists\bar{\alpha}.C \wedge \mathsf{def}\ x : \forall\bar{\alpha}[C].\tau\ \mathsf{in}\ C'$$

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

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

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

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

Figure 5.5: Constraint generation for ML

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

$$
\begin{array}{llcll}
\textit{(1)} & \exists \alpha.(\langle\!\langle a : \alpha \rangle\!\rangle \wedge \alpha = \tau) & \equiv & \langle\!\langle a : \tau \rangle\!\rangle & \textit{if } \alpha \# \tau \\
\textit{(2)} & (\!| a |\!) \le \tau & \equiv & \langle\!\langle a : \tau \rangle\!\rangle & \\
\textit{(3)} & [x \mapsto (\!| a_1 |\!)] \langle\!\langle a_2 : \tau \rangle\!\rangle & \equiv & \langle\!\langle [x \mapsto a_1] a_2 : \tau \rangle\!\rangle &
\end{array}
$$

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

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

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

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

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

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

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

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

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

5.3.3 Constraint solving by example

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

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

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

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

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

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

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

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

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

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

Hence,

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

We have used the rule:

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

The initial constraint has now been simplified down to:

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

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

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

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

The initial constraint has now been simplified down to:

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

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

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

that is, by extrusion:

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

Finally, by eliminating a few auxiliary variables:

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

We have shown the following equivalence between constraints:

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

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

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

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

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

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

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

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

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

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

```
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 outputing the result type.

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

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

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

$$\begin{aligned}
\llparenthesis\, x\, \rrparenthesis &= \forall\alpha[x \le \alpha].\,\alpha \\
\llparenthesis\, \lambda x.\, a\, \rrparenthesis &= \forall\alpha_1\alpha_2\big[\mathsf{def}\ x : \alpha_2\ \mathsf{in}\ \llparenthesis\, a\, \rrparenthesis \le \alpha_1\big].\,\alpha_2 \to \alpha_1 \\
&\ \text{if } \alpha_1, \alpha_2 \# a \\
\llparenthesis\, a_1\ a_2\, \rrparenthesis &= \forall\alpha_1\alpha_2\big[\llparenthesis\, a_1\, \rrparenthesis \le \alpha_2 \to \alpha_1 \wedge \llparenthesis\, a_2\, \rrparenthesis \le \alpha_2\big].\,\alpha_1 \\
&\ \text{if } \alpha_1, \alpha_2 \# a_1, a_2 \\
\llparenthesis\, \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2\, \rrparenthesis &= \forall\alpha\big[\mathsf{let}\ x : \llparenthesis\, a_1\, \rrparenthesis\ \mathsf{in}\ \llparenthesis\, a_2\, \rrparenthesis \le \alpha\big].\,\alpha
\end{aligned}$$

Figure 5.6: Constraint generation with principal constraint type schemes

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

5.3.4 Type reconstruction

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

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

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

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

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

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

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

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

A term in canonical form is uniquely determined up to reordering of type abstractions and type applications by the type schemes of bound program variables and of how they are instanced. We may keep track of such information during constraint resolution by keeping the binding constraints $\mathsf{def}\ x : C\ \mathsf{in}\ C$ and its derived form $\mathsf{let}\ x : C\ \mathsf{in}\ C$, and the instantiation constraints $x \preceq \tau$ of the original constraint—instead of removing them once solved. We call them *persistent constraints*. We thus forbid the removal, as well as the extrusion of persistent constraints by restricting the equivalence of constraints accordingly.

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

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

$$\mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[x \preceq \tau] \quad \equiv \quad \mathsf{def}\ x : \sigma\ \mathsf{in}\ \mathcal{R}[x \preceq \tau \wedge \sigma \preceq \tau]$$

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

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

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

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

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

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

$$\begin{array}{rl} & \exists\gamma_2.(assoc \preceq \gamma_2 \to \gamma_1 \wedge x \preceq \gamma_2)\\ \equiv & \exists\gamma_2.(assoc \preceq \gamma_2 \to \gamma_1 \wedge x \preceq \gamma_2 \wedge \alpha_0 \preceq \gamma_2) \quad \equiv \quad assoc \preceq \alpha_0 \to \gamma_1 \wedge x \preceq \alpha_0\end{array}$$

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

$$\begin{array}{rl} & assoc \preceq \alpha_0 \to \gamma_1 \wedge \forall\alpha\beta.\,\alpha \to \mathsf{list}\ (\alpha \times \beta) \to \beta \preceq \alpha_0 \to \gamma_1 \wedge x \preceq \alpha_0\\ \equiv & assoc \preceq \alpha_0 \to \gamma_1 \wedge \exists\alpha\beta.(\alpha = \alpha_0 \wedge \mathsf{list}\ (\alpha \times \beta) \to \beta = \gamma_1) \wedge x \preceq \alpha_0\\ \equiv & \exists\beta.(assoc \preceq \alpha_0 \to \mathsf{list}\ (\alpha_0 \times \beta) \to \beta \wedge \mathsf{list}\ (\alpha_0 \times \beta) \to \beta = \gamma_1 \wedge x \preceq \alpha_0)\end{array}$$

Hence, the type scheme of assoc is equivalent to

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Type annotations

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

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

Typing and constraint generation are obvious:

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

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

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

5.4.1 Explicit binding of type variables

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

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

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

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

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

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

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

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

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

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

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

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

Constraint generation for the existential form is straightforward:

$$\langle\!\langle (\exists\bar{\alpha}.a) : \tau \rangle\!\rangle \ = \ \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \# \tau$$

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

For example, the expression $\lambda x_1.\, \lambda x_2.\, \exists\alpha.((x_1 : \alpha), (x_2 : \alpha))$ has principal type scheme $\forall\alpha.\, \alpha \to \alpha \to \alpha\times\alpha$. Indeed, the generated constraint is of the form $\exists\alpha.(\langle\!\langle x_1 : \alpha \rangle\!\rangle \wedge \langle\!\langle x_2 : \alpha \rangle\!\rangle \wedge \ldots)$, which requires x_1 and x_2 to *share* a common (unspecified) type.

Perhaps surprisingly, constraint generation for the universal case is more difficult. A term a has type scheme, say, $\forall\alpha.\, \alpha \to \alpha$ if and only if a has type $\alpha \to \alpha$ *for every instance of*

α, or, equivalently, for an abstract α. To express this in terms of constraints, we introduce *universal quantification* in the constraint language:

$$C ::= \ldots \mid \forall\alpha.C$$

Its interpretation is as expected:

$$\frac{\forall \mathbf{t}, \quad \phi[\alpha \mapsto \mathbf{t}], \psi \vdash C}{\phi, \psi \vdash \forall\alpha.C}$$

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

The need for universal quantification in constraints arises when polymorphism is *required* by the programmer, as opposed to *inferred* by the system. Constraint generation for the universal form is somewhat subtle. A naive definition *fails*:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \mathrel{\#} \tau \qquad \textit{Wrong!}$$

This requires τ to be simultaneously equal to *all* of the types that a assumes when $\bar{\alpha}$ varies. For instance, with this incorrect definition, one would have:

$$\begin{array}{ll} & \langle\!\langle \forall\alpha.(\lambda x.x : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ = & \forall\alpha.\langle\!\langle (\lambda x.x : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \equiv & \forall\alpha.(\langle\!\langle \lambda x.x : \alpha \to \alpha \rangle\!\rangle \wedge \alpha = \mathsf{int}) \quad \equiv \quad \forall\alpha.(\mathsf{true} \wedge \alpha = \mathsf{int}) \quad \equiv \quad \mathsf{false} \end{array}$$

A correct definition is:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\exists\gamma.\langle\!\langle a : \gamma \rangle\!\rangle \wedge \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle$$

This requires a to be well-typed *for all* instances of $\bar{\alpha}$ and requires τ to be a valid type for a under *some* instance of $\bar{\alpha}$.

However, a problem with this definition is that the term a is duplicated, which can lead to exponential complexity. Fortunately, this can be avoided modulo a slight extension of the constraint language (Pottier and Rémy, 2003, p. 112). The solution defines:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \mathsf{let}\ x : \forall\vec{\alpha}, \beta[\langle\!\langle a : \beta \rangle\!\rangle].\,\beta\ \mathsf{in}\ x \preceq \tau$$

where the new constrain form satisfies the equivalence:

$$\mathsf{let}\ x : \forall\vec{\alpha}, \vec{\beta}[C_1].\,\tau\ \mathsf{in}\ C2 \equiv \forall\vec{\alpha}.\,\exists\vec{\beta}.\,C_1 \wedge \mathsf{def}\ x : \forall\vec{\alpha}, \vec{\beta}[C_1].\,\tau\ \mathsf{in}\ C2$$

Annotating a term with a *type scheme,* rather than just a type, is now just syntactic sugar:

$$(a : \forall\bar{\alpha}.\tau) \quad \triangleq \quad \forall\bar{\alpha}.(a : \tau) \qquad \text{if } \bar{\alpha} \mathrel{\#} a$$

In that particular case, constraint generation is in fact simpler:

$$\langle\!\langle (a : \forall\bar{\alpha}.\tau) : \tau' \rangle\!\rangle \quad \equiv \quad \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \wedge (\forall\bar{\alpha}.\tau) \preceq \tau'$$

Exercise 39 *Check this equivalence.* □

Examples Consider the following two examples:

$$\begin{array}{rl} & \langle\!\langle (\exists\alpha.(\lambda x.\, x+1 : \alpha \to \alpha)) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \equiv & \exists\alpha.\langle\!\langle (\lambda x.\, x+1 : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \equiv & \exists\alpha.(\alpha = \mathsf{int}) \\ \equiv & \mathsf{true} \end{array}$$

$$\begin{array}{rl} & \langle\!\langle (\forall\alpha.(\lambda x.\, x+1 : \alpha \to \alpha)) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \Vdash & \forall\alpha.\exists\gamma.\langle\!\langle (\lambda x.\, x+1 : \alpha \to \alpha) : \gamma \rangle\!\rangle \\ \equiv & \forall\alpha.\exists\gamma.(\alpha = \mathsf{int} \wedge \alpha \to \alpha = \gamma) \\ \equiv & \forall\alpha.\alpha = \mathsf{int} \\ \equiv & \mathsf{false} \end{array}$$

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

$$\begin{array}{rl} & \langle\!\langle (\forall\alpha.(\lambda x.\, x : \alpha \to \alpha)) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ = & \forall\alpha.\exists\gamma.\langle\!\langle (\lambda x.\, x : \alpha \to \alpha) : \gamma \rangle\!\rangle \wedge \exists\alpha.\langle\!\langle (\lambda x.\, x : \alpha \to \alpha) : \mathsf{int} \to \mathsf{int} \rangle\!\rangle \\ \equiv & \forall\alpha.\exists\gamma.\alpha \to \alpha = \gamma \wedge \exists\alpha.\alpha = \mathsf{int} \\ \equiv & \mathsf{true} \end{array}$$

The system *checks* that α is used in an abstract way, which is indeed the case here. It also checks that, if α is appropriately instantiated, the code admits the expected type $\mathsf{int} \to \mathsf{int}$.

The two next examples are similar and show the importance of the scope of existential variables. In the first one, the variable α is bound *outside* the let construct;

$$\begin{array}{rl} & \langle\!\langle \exists\alpha.(\mathsf{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true})) : \gamma \rangle\!\rangle \\ \equiv & \exists\alpha.(\mathsf{let}\ f : \alpha \to \alpha\ \mathsf{in}\ \exists\gamma_1\gamma_2.(f \le \mathsf{int} \to \gamma_1 \wedge f \le \mathsf{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma)) \\ \equiv & \exists\alpha\gamma_1\gamma_2.(\alpha \to \alpha = \mathsf{int} \to \gamma_1 \wedge \alpha \to \alpha = \mathsf{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\ \Vdash & \exists\alpha.(\alpha = \mathsf{int} \wedge \alpha = \mathsf{bool}) \\ \equiv & \mathsf{false} \end{array}$$

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

$$\begin{array}{rl} & \langle\!\langle \mathsf{let}\ f = \exists\alpha.(\lambda x.\, x : \alpha \to \alpha)\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true}) : \gamma \rangle\!\rangle \\ \equiv & \mathsf{let}\ f : \forall\beta[\exists\alpha.(\alpha \to \alpha = \beta)].\,\beta\ \mathsf{in}\ \exists\gamma_1\gamma_2.(f \le \mathsf{int} \to \gamma_1 \wedge f \le \mathsf{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\ \equiv & \mathsf{let}\ f : \forall\alpha.\,\alpha \to \alpha\ \mathsf{in}\ \exists\gamma_1\gamma_2.(\ldots) \\ \equiv & \exists\gamma_1\gamma_2.(\mathsf{int} = \gamma_1 \wedge \mathsf{bool} = \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\ \equiv & \mathsf{int} \times \mathsf{bool} = \gamma \end{array}$$

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

Type annotations in the real world For historical reasons, type variables are not explicitly bound in OCaml. (Retrospectively, that's *bad!*) They are implicitly *existentially* bound at the nearest enclosing toplevel let construct. In Standard ML, type variables are implicitly *universally* bound at the nearest enclosing toplevel let construct. In Glasgow Haskell, type variables are implicitly existentially bound within patterns: *'A pattern type signature brings into scope any type variables free in the signature that are not already in scope'* Peyton Jones and Shields (2004). Constraints help understand these varied design choices uniformly.

5.4.2 Polymorphic recursion

Recall below the typing rule FixAbs for recursive functions, which leads to the derived typing LetRec for recursive definitions:

$$\textsc{FixAbs}\ \frac{\Gamma, f : \tau \vdash \lambda x.\, a : \tau}{\Gamma \vdash \mu f.\lambda x.a : \tau} \qquad \textsc{LetRec}\ \frac{\Gamma, f : \tau_1 \vdash \lambda x.\, a_1 : \tau_1 \qquad \bar{\alpha} \mathrel{\#} \Gamma, a_1 \qquad \Gamma, f : \forall \bar{\alpha}.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let\ rec}\ f\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}$$

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

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

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

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

$$\textsc{FixAbsPoly}\ \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu f.\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\ \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ f\ x = a_1\ \mathsf{in}\ a_2 : \tau}$$

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

In System F, there is no problem to begin with; no extension is necessary. Polymorphic recursion alters, to some extent, Damas and Milner's type system. Now, not only *let-bound*, but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus. This has two noticeable consequences: *monomorphization,* a technique employed in some ML compilers Tolmach and Oliva (1998); Cejtin et al. (2007), is no longer possible; besides, *type inference* becomes problematic!

Type inference for ML with polymorphic recursion is undecidable Henglein (1993). It is equivalent to the undecidable problem of *semi-unification.* Yet, type inference in the presence of polymorphic recursion can be made simple by relying on a *mandatory type annotation.*

The syntax and typing rules for recursive definitions become:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu(f : \sigma).\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \mathsf{let\ rec}\ (f : \sigma) = \lambda x.\, a_1\ \mathsf{in}\ a_2 : \tau}$$

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

$$\langle\!\langle \mathsf{let\ rec}\ (f : \sigma) = \lambda x.\, a_1\ \mathsf{in}\ a_2 : \tau \rangle\!\rangle \quad = \quad \mathsf{let}\ f : \sigma\ \mathsf{in}\ (\langle\!\langle \lambda x.\, a_1 : \sigma \rangle\!\rangle \wedge \langle\!\langle a_2 : \tau \rangle\!\rangle)$$

It is clear that f receives type scheme σ both *inside and outside* of the recursive definition.

5.4.3 Unification under a mixed prefix

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

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

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

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

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

- Otherwise, one can decompose C as $\exists\bar{\beta}.(C_1 \wedge C_2)$, where $\bar{\alpha}\bar{\beta} \# C_1$ and $\exists\bar{\beta}.C_2 \equiv \mathsf{true}$; in that case, $\forall\bar{\alpha}.C$ reduces to just C_1.

  For example, $\forall\alpha.\exists\beta\gamma_1\gamma_2.(\beta = \alpha \to \gamma \wedge \gamma = \gamma_1 \to \gamma_2)$ reduces to just $\exists\gamma_1\gamma_2.(\gamma = \gamma_1 \to \gamma_2)$. The constraint $\forall\alpha.\exists\beta.(\beta = \alpha \to \gamma)$ is equivalent to true.

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

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

Values do not match: val id : int → int
is not included in val id : 'a → 'a
```

This gives rise to a constraint of the form $\forall\alpha.\alpha = \mathsf{int}$, while the following example gives rise to a constraint of the form $\exists\beta.\forall\alpha.\alpha = \beta$:

```
let r = ref (fun x → x) in
let module M : sig val id : 'a → 'a end = struct let id = !r end in M.id;;
```

```
Values do not match: val id : '_a → '_a
is not included in val id : 'a → 'a
```

5.5 Equi- and iso-recursive types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees. Indeed, if the grammar of types is $\tau ::= \mathsf{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values. For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not: “A list is either empty or a pair of an element and a list.” We need something like this:

$$\mathsf{list}\ \alpha \quad \diamond \quad \mathsf{unit} + \alpha \times \mathsf{list}\ \alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches. In the equi-recursive approach, a recursive type is *equal* to its unfolding. In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions.*

5.5.1 Equi-recursive types

In the equi-recursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid F\ \vec{\tau}$$

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

$$\tau ::= \alpha \mid \mu\alpha.(F\ \vec{\tau})$$

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

$$\mu\beta.(\mathsf{unit} + \alpha \times \beta)$$

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

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

One can also prove Brandt and Henglein (1998) that equality is the least congruence generated by the following two rules:

$$\textsc{Fold/Unfold}\quad \mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau \qquad\qquad \textsc{Uniqueness}\quad \frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

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

Exercise 40 *Prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from* `Inductive` *to* `CoInductive`*.* □

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

Exercise 41 *Describe solved forms and show that every solved form is either* false *or satisfiable.* □

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

$$\mu(\mathit{length}).\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \lambda().\,0 \diamond \lambda(y,\,z).\,1 + \mathit{length}\ z$$

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

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

```
type ('a, 'b) sum = Left of 'a | Right of 'b
  type ('a, 'b) sum = Left of 'a | Right of 'b

let rec length x = function Left () → 0 | Right (y, z) → 1 + length z
val length : ((unit, 'b * 'a) sum as 'a) → int = ⟨fun⟩
```

Notice that `-rectypes` is only an option which is not on by default. Equi-recursive types are simple and powerful, but in practice, they are perhaps *too expressive*. Continuing with in the `-rectype` option:

```
let rec map f = function [] → [] | y :: z → map f y :: map f z
val map : 'a → ('b list as 'b) → ('c  list  as 'c) = ⟨fun⟩

map (fun x → x + 1) [ 1; 2 ]
This expression has type int but is used with type 'a list as 'a

map () [[];  [[]]]
– : 'a  list  as 'a = [[];  [[]]]
```

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error. Hence, by default, OCaml typechecker reject type cycles that do not involve an object type or a variant type. In a normal OCaml session (no `-rectypes`), the following is still accepted, though:

```
let f x = x#hello x;;
val f : (< hello : 'a → 'b;  ..  > as 'a) → 'b = ⟨fun⟩
```

OCaml implements a partial occurs check that stops at object and variant types: equi-recursive types are allowed provided every infinite path crosses an object or a variant type.

5.5.2 Iso-recursive types

In the iso-recursive approach, the user is allowed to introduce new *type constructors* D via (possibly mutually recursive) *declarations:*

$$D\,\vec{\alpha} \approx \tau \qquad (\text{where } \mathsf{ftv}(\tau) \subseteq \vec{\alpha})$$

Each such declaration adds a unary constructor fold_D and a unary destructor unfold_D with the following types and the new reduction rule:

$$\mathsf{fold}_D \;:\; \forall\bar{\alpha}.\,\tau \to D\,\vec{\alpha} \qquad\qquad \mathsf{unfold}_D \;:\; \forall\bar{\alpha}.\,D\,\vec{\alpha} \to \tau \qquad\qquad \mathsf{unfold}_D\;(\mathsf{fold}_D\;v) \longrightarrow v$$

Ideally, iso-recursive types should not have any runtime cost. One solution is to compile constructors and destructors away into a target language with equi-recursive types. Another solution is to see iso-recursive types as a restriction of equi-recursive types where the source language does not have equi-recursive types but instead two unary destructors fold_D and unfold_D with the semantics of the identity function. Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the Uniqueness typing rule.

For, example iso-recursive lists can be defined as follows. A parametrized, iso-recursive type of lists is: $\mathsf{list}\ \alpha \approx \mathsf{unit} + \alpha \times \mathsf{list}\ \alpha$. The empty list is: $\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ())\ :\ \forall\alpha.\,\mathsf{list}\ \alpha$. A function that measures the length of a list is:

$$\mu(\mathit{length}).\lambda xs.\mathsf{case}\ (\mathsf{unfold}_{\mathsf{list}}\ xs)\ \mathsf{of}\ \lambda().\,0 \diamond \lambda(x, xs).\,1 + \mathit{length}\,xs\ \ :\ \ \forall\alpha.\,\mathsf{list}\ \alpha \rightarrow \mathsf{int}$$

One *folds upon construction* and *unfolds upon deconstruction.*

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

5.5.3 Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums. This suppresses the *verbosity* of explicit folds and unfolds as well as the *fragility* and inconvenience of numeric indices—instead, named *record fields* and *data constructors* are used. For instance,

$$\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ())\quad \text{is replaced with}\quad \mathit{Nil}\,()$$

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

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

The set L denotes a finite set of record labels or data constructors $\{\ell_1 \ldots \ell_n\}$, which is fixed for a given definition. Algebraic data type definitions can be mutually recursive.

The record type definition $D\,\vec{\alpha} \approx \prod_{\ell\in L} \ell : \tau_\ell$ introduces a record n-ary *constructor* and n record unary *destructors* with the following types:

$$C ::= \ldots \mid \{\ell_1 = \cdot\,, \ldots \ell_n = \cdot\,\} \qquad\qquad d ::= \ldots \mid (\,\cdot\,.\ell_1) \mid \ldots (\,\cdot\,.\ell_n)$$

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

The variant type definition $D\,\vec{\alpha} \approx \sum_{\ell\in L} \ell : \tau_\ell$ introduces unary variant constructors and variant destructor of arity $n + 1$ with the following types:

$$C ::= \ldots \mid (\ell\,\cdot) \qquad d ::= \ldots \mid \mathsf{case}\ \cdot\ \mathsf{of}\ [\ell_1 : \cdot \diamond \ldots \ell_n : \cdot] \qquad \ell : \forall\vec{\alpha}.\,\tau_\ell \rightarrow D\,\vec{\alpha}$$

$$\mathsf{case}\ \cdot\ \mathsf{of}\ [\ell_1 : \cdot \diamond \ldots \ell_n : \cdot] : \forall\vec{\alpha}\beta.\,D\,\vec{\alpha} \rightarrow (\tau_{\ell_1} \rightarrow \beta) \rightarrow \ldots (\tau_{\ell_n} \rightarrow \beta) \rightarrow \beta$$

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

$$\begin{aligned} \mathsf{case}\ \cdot\ \mathsf{of}\ [\mathit{Nil} : \cdot \diamond \ldots \mathit{Cons} : \cdot] :&\quad \forall\alpha\beta.\,\mathsf{list}\ \alpha \rightarrow (\mathsf{unit} \rightarrow \beta) \rightarrow ((\alpha \times \mathsf{list}\ \alpha) \rightarrow \beta) \rightarrow \beta \\ \mathit{Nil} :&\quad \forall\alpha.\,\mathsf{unit} \rightarrow \mathsf{list}\ \alpha \\ \mathit{Cons} :&\quad \forall\alpha.\,(\alpha \times \mathsf{list}\ \alpha) \rightarrow \mathsf{list}\ \alpha \end{aligned}$$

$$\textsc{hm-Var}\quad \frac{\sigma = \Gamma(x) \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma} \qquad \textsc{hm-Abs}\quad \frac{C, (\Gamma, x : \tau_0) \vdash a : \tau}{C, \Gamma \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad \textsc{hm-App}\quad \frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{hm-Let}\quad \frac{C, \Gamma \vdash a_1 : \sigma \qquad C, (\Gamma, x : \sigma) \vdash a_2 : \tau}{C, \Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau} \qquad \textsc{hm-Gen}\quad \frac{C \wedge C_0, \Gamma \vdash a : \tau \qquad \vec\alpha \mathrel{\#} C, \Gamma}{C \wedge \exists\vec\alpha.\, C_0, \Gamma \vdash a : \forall\vec\alpha[C_0].\, \tau}$$

$$\textsc{hm-Inst}\quad \frac{C, \Gamma \vdash a : \forall\vec\alpha[C_0].\, \tau}{C \wedge C_0, \Gamma \vdash a : \tau} \qquad \textsc{hm-Sub}\quad \frac{C, \Gamma \vdash a : \tau_1 \qquad C \Vdash \tau_1 \le \tau_2}{C, \Gamma \vdash a : \tau_2} \qquad \textsc{hm-Exists}\quad \frac{C, \Gamma \vdash a : \tau \qquad \vec\alpha \mathrel{\#} \Gamma, \tau}{\exists\vec\alpha.\, C, \Gamma \vdash a : \tau}$$

Figure 5.7: Typing rules for HM(X)

A function that measures the length of a list is:

$$\mu(\mathit{length}).\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathit{Nil} : \lambda().\,0 \diamond \mathit{Cons} : \lambda(y,\, z).\,1 + \mathit{length}\ z \quad : \quad \forall\alpha.\,\mathsf{list}\ \alpha \to \mathsf{int}$$

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

5.6 HM(X)

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

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

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

The typing rules also use an entailment predicate $C \Vdash C'$ between constraints that is more general than constraint equivalence. Entailment is defined as expected: $C \Vdash C'$ if and only if any ground assignment that satisfies C also satisfies C'.

Typing rules for $\mathsf{HM(X)}$ are presented in Figure 5.7. Moreover, judgment are taken up to constraint equivalence. The constraint $\exists\sigma$ in the premise of Rule $\textsc{hm-Var}$ is an abbreviation for $\exists\bar{\alpha}.C_0$ where σ is $\forall\bar{\alpha}[C_0].\,\tau$. A valid judgment is one that has a derivation with those typing rules. In a valid judgment, C may not be satisfiable. A program is well-typed in environment Γ if it has a valid judgment $C, \Gamma \vdash a : \tau$ for some τ and *satisfiable* constraint C.

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

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

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

$$\textsc{Exists}\ \dfrac{\textsc{Let}\ \dfrac{\textsc{Fun}\ \dfrac{\overset{\textsc{Var}}{x \le \alpha \vdash x : \alpha}}{\mathsf{let}\, x : \alpha_0 \,\mathsf{in}\, x \le \alpha \vdash \lambda x.\, x : \alpha_0 \to \alpha} \qquad \textsc{App}\ \dfrac{\overset{\textsc{Var}}{y \le \beta_2 \to \beta_1 \vdash y : \beta_2 \to \beta_1} \qquad \overset{\textsc{Var}}{y \le \beta_2 \vdash y : \beta_2}}{y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \vdash y\, y : \beta_1}}{C \vdash a : \beta_1}}{\exists\beta_2.C \vdash a : \beta_1}$$

where C is

$$\mathsf{let}\, y : \forall\alpha\alpha_0[\mathsf{let}\, x : \alpha_0 \,\mathsf{in}\, x \le \alpha].\, \alpha_0 \to \alpha \,\mathsf{in}\, y \le \beta_2 \to \beta_1 \wedge y \le \beta_2$$

The constraint C can be simplified as follows:

$$\begin{array}{rcl}
\exists\beta_2.C & = & \exists\beta_2.\, \mathsf{let}\, y : \forall\alpha\alpha_0[\alpha_0 = \alpha].\, \alpha_0 \to \alpha \,\mathsf{in}\, y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \\
 & \equiv & \exists\beta_2.\, \mathsf{let}\, y : \forall\alpha.\, \alpha \to \alpha \,\mathsf{in}\, y \le \beta_2 \to \beta_1 \wedge y \le \beta_2 \\
 & \equiv & \exists\beta_2\alpha_1\alpha_2.\ \ \alpha_1 \to \alpha_1 = \beta_2 \to \beta_1 \ \ \wedge\ \ \alpha_2 \to \alpha_2 = \beta_2 \\
 & \equiv & \exists\alpha.\, \beta_1 = \alpha \to \alpha
\end{array}$$

Hence, we also have $\exists\alpha.\, \beta_1 = \alpha \to \alpha \vdash a : \beta_1$. This is a valid judgment, but not a satisfiable one. However, by rule $\textsc{pcb-Sub}$ and $\textsc{pcb-Exists}$, we have $\exists\beta_1.\, (\exists\alpha.\, \beta_1\alpha \to \alpha) \wedge \beta_1 = \beta \to \beta) \vdash a : \beta \to \beta$, which is equivalent to $\mathsf{true} \vdash a : \beta \to \beta$ and is both valid and satisfiable.

The type inference algorithm for ML is sound and complete for $\mathsf{PCB(X)}$:

- *Soundness:* $\langle\!\langle a : \tau \rangle\!\rangle \vdash a : \tau$. The constraint inferred for a typing validates the typing.

- *Completeness:* If $C \vdash a : \tau$ then $C \Vdash \langle\!\langle a : \tau \rangle\!\rangle$. The constraint inferred for a typing is more general than any constraint that validates the typing.

$$\overset{\textsc{pcb-Var}}{\frac{C \Vdash x \leq \tau}{C \vdash x : \tau}} \qquad \overset{\textsc{pcb-Abs}}{\frac{C \vdash a : \tau}{\mathsf{let}\ x : \tau_0\ \mathsf{in}\ C \vdash a : \tau_0 \to \tau}} \qquad \overset{\textsc{pcb-App}}{\frac{C_1 \vdash a_1 : \tau_2 \to \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{C_1 \wedge C_2 \vdash a_1\ a_2 : \tau_1}}$$

$$\overset{\textsc{pcb-Let}}{\frac{C_1 \vdash a_1 : \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{\mathsf{let}\ x : \forall \mathcal{V}[C_1].\, \tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}} \qquad \overset{\textsc{pcb-Sub}}{\frac{C \vdash a : \tau_1}{C \wedge \tau_1 \leq \tau_2 \vdash a : \tau_2}} \qquad \overset{\textsc{pcb-Exists}}{\frac{C \vdash a : \tau \qquad \alpha \# \tau}{\exists \alpha. C \vdash a : \tau}}$$

Figure 5.8: Typing rules for PCB(X)

Note Our presentation of HM(X) is incomplete. See also Skalka and Pottier (2002) for a more recent presentation of HM(X) and Pottier and Rémy (2005) for a detailed presentation of several variants of HM(X).

Our proof of type soundness for ML only applies for HM(=). One may prove type soundness for HM(X) in the general case for some logic X, under the axiom that the arrow type constructor is contra-variant for subtyping. See Pottier and Rémy (2005).

5.7 Type reconstruction in System F

Type checking in explicitly-typed System F is easy. Still, an implementation must carefully deal with variable bindings and renaming when applying type substitutions. However, as we have seen, programming with fully-explicit types is unpractical.

Full type inference in System F has long been an open problem, until Wells (1999) proved it undecidable by showing that it is equivalent to the semi-unification problem which was earlier proved undecidable. (Notice that the full type-inference problem is not directly related to second-order unification but rather to semi-unification.)

Hence, we must perform *partial type inference* in System F. Either type inference is incomplete, or some amount of type annotations must be provided. Several solutions are used in practice. They alleviate the need for a lot of redundant type annotations.

5.7.1 Type inference based on Second-order unification

Full type inference is equivalent to semi-unification. However, type inference becomes equivalent to second-order unification if all the positions of type abstractions and type applications are explicit, while types are themselves left implicit. That is, if terms are

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

where the question marks stand for type variables and types to be inferred. Although, the problem of second-order unification is undecidable, there are semi-algorithms that often work well in common cases. This method was proposed by Pfenning (1988).

$$\overset{\textsc{Var-I}}{\dfrac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau}} \qquad \overset{\textsc{Abs-C}}{\dfrac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau}} \qquad \overset{\textsc{App-I}}{\dfrac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\, a_2 \Uparrow \tau_1}} \qquad \overset{\textsc{I-C}}{\dfrac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau}}$$

$$\overset{\textsc{Annot-I}}{\dfrac{\Gamma \vdash a \Downarrow \tau}{\Gamma \vdash (a : \tau) \Uparrow \tau}} \qquad \overset{\textsc{Abs-I}}{\dfrac{\Gamma, x : \tau_0 \vdash a \Uparrow \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, a \Uparrow \tau_0 \to \tau}}$$

Figure 5.9: Bidirectional type checking for the simply-typed λ-calculus .

In fact, partial type inference based on second-order unification can be mixed with type checking. Explicit polymorphism may be reintroduced as in explicitly-typed System F while explicitly-controlled implicit instantiation can be performed as above by second-order unification. The source language is:

$$M ::= x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid \lambda x{:}?.\, M \mid M\ ? \mid \mathsf{let}\ f = \Lambda^? \alpha_1. \ldots \Lambda^? \alpha_n.M\ \mathsf{in}\ M$$

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

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

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

5.7.2 Bidirectional type inference

Type-checking in explicit simply-typed λ-calculus is easy because typing rules have an algorithmic reading. This implies that they are syntax directed, but also that judgments can be read as functions where some arguments are inputs and others are output. In the implicit calculus, the rules are still syntax-directed, but some of them do not have an obvious algorithmic reading. Typically, Γ and a would be inputs and τ is an output in the judgment $\Gamma \vdash a : \tau$, which we may represent as $\Gamma^\uparrow \vdash a^\uparrow : \tau^\downarrow$. However, in the rule for abstraction:

$$\overset{\textsc{Abs}}{\dfrac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}}$$

the type τ_0 is used both as input (in the premise) and as an output in the conclusion. Hence, type-checking the implicit simply-typed λ-calculus is not straightforward. In some cases, the type of the function may be known, *e.g.* when the function is an argument to an expression of a known type. Then, it suffices to check the proposed type is indeed correct.

Formally, we need algorithmic reading of the typing judgment, depending on whether the return type is known or unknown. We may split the typing judgment $\Gamma \vdash a : \tau$ into two

$$\textsc{Abs-C}\;\dfrac{\textsc{I-C}\;\dfrac{\textsc{App-I}\;\dfrac{\textsc{Var-I}\;\dfrac{}{\Gamma \vdash f \Uparrow \tau} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau_1 \vdash x \Uparrow \tau_1}\;\textsc{Var-I}}{\Gamma, x : \tau_1 \vdash x \Downarrow \tau_1}\;\textsc{C-I}}{\Gamma \vdash \lambda x.\, x \Downarrow \tau_1 \to \tau_1}\;\textsc{Abs-C}}{\Gamma \vdash f\ (\lambda x.\, x) \Uparrow \tau_2}}{\Gamma \vdash f\ (\lambda x.\, x) \Downarrow \tau_2}}{\varnothing \vdash \lambda f : \tau.\, f\ (\lambda x.\, x) \Downarrow \tau \to \tau_2}$$

Figure 5.10: Example of bidirectional derivation

judgments $\Gamma \vdash a \Downarrow \tau$ to check that a may be assigned the type τ and $\Gamma \vdash a \Uparrow \tau$ to infer the type τ of a (or with information flows $\Gamma^{\uparrow} \vdash a^{\uparrow} \Downarrow \tau^{\uparrow}$ and $\Gamma^{\uparrow} \vdash a^{\uparrow} \Uparrow \tau^{\downarrow}$. Both judgments are recursively defined by the rules of Figure **??**: the checking mode can call the inference mode when needed; conversely, annotations may be used to turn inference mode into checking mode. (As a particular case, annotations on type abstractions enable the inference mode.)

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

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

Predicative polymorphism *Predicative polymorphism* is an interesting subcase of bidirectional type inference in the presence of predicative polymorphism. Predicative polymorphism is a restriction of impredicative polymorphism as can be found in System F. With predicative polymorphism, types are stratified so that polymorphic types can only be instantiated with simple types.

Interestingly, partial type inference can then still reduced to typing constraints under a mixed prefix (Rémy, 2005; Jones et al., 2006). Unfortunately, predicative polymorphism is too restrictive for use in programming languages: as polymorphic values often need to be put in data-structures whose constructors are polymorphic but impredicative polymorphism does not allow implicit instantiation of polymorphic constructors by polymorphic types.

One may also use a hierarchy of types where polymorphic types of rank n can be instantiated with polymorphic types of a strictly lower rank. This increases expressiveness but F is still more expressive than the union of all F^n.

Type inference with first-order constraints does not work for higher ranks.

Local type inference A simpler approach than *global* bidirectional type inference proposed by Pierce and Turner and improved by Odersky et al. is to perform bidirectional type inference *locally*, *i.e.* by considering for each node only a small context surrounding it.

Subtyping Interestingly, bidirectional type inference can easily be extended to work in the presence of subtyping, which is not the case for methods based on second order unification.

5.7.3 Partial type inference in MLF

The language MLF (Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008) is an extension of System F especially designed for partial type inference—in fact for type inference a la ML within System F. That is, the inference algorithm performs first-order unification and aggressive ML-style let-generalization, but in the presence of second-order types. Interestingly, only parameters of functions that are used polymorphically need to be annotated in MLF; type abstractions and type annotation are always left implicit. However, for the purpose of type inference, MLF introduces richer types that enable to write "more principal types", but that are also harder to read. The type inference method for MLF can be seen as a generalization of the constraint-based type inference for ML that handles polymorphic types.

5.8 Proofs and Solution to Exercises

Proof of Theorem 15

We proof $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ if and only if $\phi\Gamma \vdash a : \phi\tau$ by induction on a. We prove both implications independently because reasoning with equivalence is error-prone, since the arguments are similar but often not quite the same in both directions. The proof is thus a bit lengthy, but all cases are easy.

Case a is x: Assume $\phi\Gamma \vdash a : \phi\tau$. By inversion of typing, this judgment must be derived by rule Var. Hence, $\phi\tau = \phi\Gamma(x)$. By definition of satisfiability this implies $\phi \vdash \tau = \Gamma(x)$. By definition of typing constraint, this is $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

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

Case a is $a_1\ a_2$: Assume $\phi\Gamma \vdash a : \phi\tau$. By rule App, there exists τ_2 such that $\phi\Gamma \vdash a_1 : \tau_2 \rightarrow \phi\tau$ and $\phi\Gamma \vdash a_2 : \tau_2$. Let $\beta \# \Gamma$ and ϕ' be $\phi, \beta \mapsto \tau_2$. We have $\phi'\Gamma \vdash a_1 : \phi'\beta \rightarrow \tau$ and $\phi'\Gamma \vdash a_2 : \beta$. Hence, by induction hypothesis $\phi' \vdash \langle\!\langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle\!\rangle$ and $\phi' \vdash \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle$. Thus, $\phi \vdash \exists \beta. \langle\!\langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle$. *i.e.* $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

Conversely, assume $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$. We have $\phi \vdash \exists \beta. \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle\!\rangle$. We may assume *w.l.o.g.* that $\beta \# \phi$. There must exist ϕ' of the form $\phi, \beta \mapsto \tau_2$ such that $\phi' \vdash \langle\!\langle \Gamma \vdash a_2 : \beta \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_1 : \beta \rightarrow \tau \rangle\!\rangle$. 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 App, we have $\phi\Gamma \vdash a_1\ a_2 : \phi\tau$.

Case a is $\lambda x.\, a_1$: Assume $\phi\Gamma \vdash a : \phi\tau$. We may assume *w.l.o.g.* that $x \# \Gamma$. By rule Fun, there must exist τ_1 and τ_2 such that $\phi\Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi\tau = \tau_2 \to \tau_1$. Let β_1 and β_2 be disjoint from Γ and ϕ' be $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$. Then, both $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1$ and $\phi'\tau = \phi'(\beta_2 \to \beta_1)$ hold. By induction hypothesis, $\phi' \vdash \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \tau_1 \rangle\!\rangle$ and $\phi' \vdash \tau = \beta_2 \to \beta_1$ Therefore, $\phi \vdash \exists \beta_1 \beta_2. \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle\!\rangle \wedge \tau = \beta_2 \to \beta_1$. That is, $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$.

Conversely, assume $\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$. By definition of constraints, we have $\phi \vdash \exists \beta_1 \beta_2. \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \beta_1 \rangle\!\rangle \wedge \tau = \beta_2 \to \beta_1$ for some x disjoint from Γ. We may assume *w.l.o.g.* that $\beta_1, \beta_2 \# \phi$. There must exist ϕ' of the form $\phi, \beta_2 \mapsto \tau_2, \beta_1 \mapsto \tau_1$ such that $\phi' \vdash \langle\!\langle \Gamma, x : \beta_2 \vdash a_1 : \tau_1 \rangle\!\rangle$ and $\phi' \vdash \tau = \beta_2 \to \beta_1$. By induction hypothesis, $\phi'(\Gamma, x : \beta_2) \vdash a_1 : \phi'\beta_1$ and $\phi'\tau = \phi'(\beta_2 \to \beta_1)$. That is, $\phi\Gamma, x : \tau_2 \vdash a_1 : \tau_1$ and $\phi\tau = \tau_2 \to \tau_1$. Hence, by rule Fun, we have $\phi\Gamma \vdash a : \phi\tau$.

Solution of Exercise 36

See Bjørner (1994). ■

Solution of Exercise 37

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

François Pottier. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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